Machine Learning (2EL1730)

Complete Course Notes

Based on lectures by Nora Ouzir

2025–2026

Contents

Lecture 1: Introduction, Model Selection, and Evaluation **10**

1 Introduction to Machine Learning **10**
1.1 Definition . . . 10
1.2 Why Learning? . . . 10
1.3 Basic Machine Learning Pipeline . . . 10

2 Types of Machine Learning **11**
2.1 Unsupervised Learning . . . 11
2.2 Supervised Learning . . . 11

3 Formalization of Supervised Learning **11**
3.1 The Setup . . . 11
3.2 Building a Model: 3 Steps . . . 12
3.3 Loss Functions . . . 12
3.3.1 For Classification ($y \in \{-1, +1\}$) . . . 12
3.3.2 For Regression ($y \in \mathbb{R}$) . . . 12
3.4 Empirical Risk Minimization (ERM) . . . 12

4 Optimization **13**
4.1 Convex Optimization . . . 13
4.1.1 Definitions . . . 13
4.1.2 Characterizations of Convex Functions . . . 13
4.2 Gradient Descent . . . 13

5 Model Selection and Generalization **14**
5.1 Overfitting vs. Underfitting . . . 14
5.2 Bias-Variance Decomposition . . . 14
5.3 Cross-Validation . . . 15

6 Evaluation Metrics **16**
6.1 Classification Metrics . . . 16
6.2 ROC Curve . . . 16

Lecture 2: Linear Regression **18**

7 Introduction to Regression **19**
7.1 Goal and Applications . . . 19

8 Linear Regression **19**
8.1 Problem Formulation . . . 19
8.2 Loss Function: Least Squares . . . 20
8.3 Optimization . . . 20
8.3.1 1. Gradient Descent (Batch GD) . . . 20
8.3.2 2. Stochastic Gradient Descent (SGD) . . . 21
8.3.3 3. Closed-Form Solution (Normal Equation) . . . 21

9 Model Complexity and Generalization **21**
9.1 The Bias-Variance Tradeoff . . . 22
9.1.1 Derivation of the Decomposition . . . 22
9.2 Regularization . . . 23

10 Classification as Regression? **23**
10.1 Decision Rule . . . 23
10.2 Limitations . . . 24
10.2.1 1. Unbounded Predictions . . . 24
10.2.2 2. Sensitivity to Outliers . . . 25

11 Logistic Regression **25**
11.1 The Sigmoid Function . . . 26
11.2 Decision Boundary . . . 26
11.3 Likelihood and Cost Function . . . 26
11.3.1 Probabilistic Interpretation . . . 26
11.3.2 The Likelihood Function . . . 27
11.3.3 Log-Likelihood . . . 27
11.3.4 The Cost Function: Binary Cross-Entropy . . . 27
11.4 Gradient Descent for Logistic Regression . . . 27

12 Lab 1: Ridge and Logistic Regression **28**
12.1 Ridge Regression Theory . . . 28
12.1.1 1. Problem Formulation . . . 28
12.1.2 2. Closed-Form Solution . . . 28
12.1.3 3. Hessian Matrix . . . 28
12.2 Python Implementation: Logistic Regression . . . 28
12.2.1 1. Core Functions . . . 28
12.2.2 2. Optimization Pipeline . . . 29

Lecture 3: Linear Classification **30**

13 Introduction and Recap **31**
13.1 Linear Methods Recap . . . 31

14 Logistic Regression **31**
14.1 From Regression to Classification . . . 31
14.2 Probabilistic Interpretation . . . 32
14.3 Maximum Likelihood Estimation (MLE) . . . 32
14.3.1 Gradient Ascent . . . 33
14.4 Decision Boundary . . . 33

15 Discriminative vs. Generative Models **33**

16 Gaussian Discriminant Analysis (GDA) **34**
16.1 The Model . . . 35
16.2 Parameter Estimation (MLE) . . . 35
16.3 Classification . . . 35

17 Naïve Bayes **36**
17.1 The Naïve Assumption . . . 36
17.2 The Classifier . . . 36
17.3 Example: Spam Classification . . . 37
17.3.1 Parameter Estimation . . . 37
17.3.2 Laplace Smoothing . . . 37

18 Fisher Linear Discriminant Analysis (FDA) **37**
18.1 The Goal . . . 38
18.2 Mathematical Formulation . . . 38
18.2.1 Projected Means . . . 38
18.2.2 Projected Variances . . . 39
18.3 Fisher's Criterion . . . 39
18.3.1 Optimization Derivation . . . 40
18.3.2 Solving for w . . . 40
18.4 Algorithm Summary . . . 41

19 Lab 2: Discriminant Analysis **41**
19.1 Naive Bayes Exercise . . . 41
19.2 FDA Implementation . . . 41
19.2.1 Algorithm Steps . . . 41
19.2.2 Python Pipeline . . . 42

Lecture 4: Support Vector Machines **43**

20 Introduction and Recap **43**
20.1 Supervised Learning Review . . . 43
20.2 Generative vs. Discriminative Models . . . 43

21 Linear SVM: Hard-Margin **44**
21.1 The Linearly Separable Case . . . 44
21.2 Geometry of the Margin . . . 45
21.3 Problem Formulation . . . 46

22 Optimization Framework **47**
22.1 Lagrange Multipliers and the Dual Problem . . . 47
22.1.1 Intuition: Forces and Prices . . . 48
22.2 Deriving the Dual . . . 48
22.3 Support Vectors and Complementary Slackness . . . 49

23 Soft-Margin SVM: The Non-Separable Case **49**
23.1 Motivation . . . 49
23.2 Slack Variables . . . 50
23.3 Soft-Margin Optimization (Primal) . . . 50
23.4 Hinge Loss Interpretation . . . 51
23.5 Soft-Margin Dual Problem . . . 51

24 Kernel Methods: The Non-Linear Case **52**
24.1 The Need for Non-Linearity . . . 52
24.2 Feature Mapping . . . 52
24.3 The Kernel Trick . . . 53
24.4 Common Kernels . . . 54
24.4.1 Polynomial Kernel . . . 54
24.4.2 Radial Basis Function (RBF) / Gaussian Kernel . . . 54

25 Lab Work Summary **55**
25.1 SVM with Linear Kernel . . . 55
25.2 SVM with Gaussian Kernel . . . 55
25.3 Text Categorization (Sentiment Analysis) . . . 56

26 Summary **56**

Lecture 5: Neural Networks **57**

27 Introduction **58**

28 Review: Support Vector Machines (SVMs) **58**
28.1 Discriminative vs. Generative Models . . . 58
28.2 Hard Magnitude and Soft Margin SVMs . . . 59
28.3 The Kernel Trick . . . 60
28.3.1 Common Kernels . . . 61
28.3.2 Effect of Kernel Parameters . . . 61

29 Dimensionality Reduction **62**
29.1 Motivation: The Curse of Dimensionality . . . 62
29.2 Intrinsic Dimensionality and Rank . . . 62
29.3 Signal vs. Noise . . . 62

30 Singular Value Decomposition (SVD) **63**
30.1 Definition . . . 63
30.2 Low-Rank Approximation . . . 64
30.3 Choosing the Rank k . . . 64

31 Principal Component Analysis (PCA) **65**
31.1 Standardization . . . 65
31.2 Variance Maximization: The Derivation . . . 65
31.3 The PCA Algorithm . . . 67
31.4 Choosing Dimensions . . . 68

32 Non-Linear Dimensionality Reduction **68**
32.1 Multidimensional Scaling (MDS) . . . 68
32.2 Isomap . . . 68
32.3 Locally Linear Embedding (LLE) . . . 69

33 Feature Selection **70**
33.1 Techniques . . . 70

34 Lab Session: Dimensionality Reduction **71**
34.1 Image Compression using SVD . . . 71
34.2 PCA on the Wine Dataset . . . 71
34.3 Map Reconstruction with MDS . . . 72

35 Conclusion **74**

Lecture 6: Decision Trees and Ensemble Methods **75**

36 Introduction **76**

37 Unsupervised Learning **76**
37.1 Supervised vs. Unsupervised . . . 76

38 Clustering Fundamentals **76**
38.1 What is Clustering? . . . 76
38.2 Applications . . . 77
38.3 Evaluation Methodology . . . 77
38.3.1 Ambiguity . . . 77
38.3.2 Quality Criteria . . . 78

39 K-Means Clustering **78**
39.1 The Algorithm (Lloyd's Algorithm) . . . 78
39.2 Objective Function and Convergence . . . 79
39.3 Voronoi Tessellation . . . 79
39.4 Initialization and K-Means++ . . . 79
39.4.1 K-Means++ Initialization . . . 80
39.5 Choosing K . . . 81

40 Spectral Clustering **81**
40.1 Motivation: Non-Convex Clusters . . . 81
40.2 Graph-Based Perspective . . . 82
40.2.1 Similarity Graph Construction . . . 82
40.3 The Graph Laplacian . . . 82
40.3.1 Properties of the Laplacian . . . 83
40.4 The Spectral Clustering Algorithm . . . 83

40.4.1 Why it works . . . 84

41 Summary: K-Means vs. Spectral **84**

42 Lab Session: Clustering Implementation **84**
42.1 K-Means on Synthetic Data . . . 84
42.2 K-Means on MNIST (Reduced) . . . 85
42.3 Spectral Clustering vs. K-Means on Non-Convex Data . . . 85

Lecture 7: Dimensionality Reduction **87**

43 Introduction **88**
43.1 Parametric vs. Non-Parametric Learning . . . 88
43.1.1 Parametric Models . . . 88
43.1.2 Non-Parametric Models . . . 88
43.2 Instance-Based Learning . . . 88

44 Similarity and Distance Measures **89**
44.1 Continuous Features . . . 89
44.1.1 Euclidean Distance (L_2 Norm) . . . 89
44.1.2 Manhattan Distance (L_1 Norm) . . . 89
44.1.3 Correlation and Cosine Similarity . . . 90
44.2 Categorical and Binary Features . . . 90
44.2.1 Hamming Distance . . . 90
44.2.2 Jaccard Similarity . . . 90

45 The K-Nearest Neighbours (K-NN) Algorithm **91**
45.1 Algorithm Definition . . . 91
45.2 Effect of K . . . 91
45.3 Choosing K . . . 91
45.4 Decision Boundaries and Voronoi Tessellations . . . 92

46 Computational Efficiency and K-D Trees **92**
46.1 The Problem: Distance Calculation Cost . . . 92
46.2 Intuition: The "Binary Search" of Multidimensional Space . . . 92
46.3 K-D Tree Construction (Building the Index) . . . 92
46.4 K-NN Search with Pruning . . . 93
46.4.1 Algorithm Steps . . . 93
46.4.2 Complexity . . . 94
46.5 The Curse of Dimensionality . . . 94

47 Variants and Practical Considerations **94**
47.1 Feature Normalization . . . 94
47.2 Weighted K-NN . . . 94

48 Lab Session: Handwritten Digit Recognition **94**
48.1 Dataset Overview . . . 95
48.2 Implementation Details . . . 95
48.3 Observations . . . 96

Lecture 8: Clustering **97**

49 Introduction **98**

50 Accelerating K-NN: K-D Trees **98**
50.1 Concept: Multidimensional Binary Search . . . 98
50.2 Tree Construction Algorithm . . . 99
50.3 Nearest Neighbour Search . . . 100
50.4 Complexity and the Curse of Dimensionality . . . 100

51 Decision Trees **101**
51.1 Model Structure . . . 101
51.2 Training: Hunt's Algorithm . . . 101
51.3 Splitting Criteria . . . 102
51.3.1 Gini Index . . . 102
51.3.2 Entropy and Information Gain . . . 102
51.4 Handling Attribute Types . . . 103
51.5 Overfitting and Pruning . . . 103

52 Ensemble Learning **104**
52.1 Bagging (Bootstrap Aggregation) . . . 104
52.1.1 Algorithm . . . 104
52.1.2 Random Forest . . . 104
52.2 Boosting . . . 105
52.2.1 Concepts . . . 105
52.2.2 Conceptual Walkthrough . . . 105
52.2.3 Key Components of AdaBoost . . . 105
52.2.4 AdaBoost Algorithm . . . 106
52.3 Comparison: Bagging vs Boosting . . . 106

53 Lab Session: AdaBoost Implementation **107**
53.1 Dataset . . . 107
53.2 Implementation Details . . . 107
53.2.1 Key Code Snippet . . . 107
53.3 Optimization and Results . . . 108

Lecture 9: Unsupervised Learning - Other Topics **109**

54 Introduction **111**

55 Recap: Ensemble Learning **111**
55.1 Bagging vs. Boosting . . . 111
55.2 AdaBoost (Adaptive Boosting) . . . 111
55.2.1 Algorithm Overview . . . 112
55.2.2 Intuition . . . 112

56 The Perceptron **113**
56.1 Model Definition . . . 113
56.2 Activation Functions . . . 114
56.3 Loss Functions . . . 114

56.4 Training: The Perceptron Learning Algorithm . . . 115
56.4.1 Theoretical Derivation (Perceptron Criterion) . . . 115
56.5 Limitations: The XOR Problem . . . 115

57 Multi-Layer Perceptron (MLP) 115
57.1 Architecture . . . 116
57.2 Solving XOR with MLP . . . 116
57.3 Universal Approximation Theorem . . . 116

58 Backpropagation 117
58.1 Problem Setup . . . 117
58.2 Derivation of Backpropagation . . . 117
58.2.1 1. Gradient for Output Weights (v_h) . . . 118
58.2.2 2. Gradient for Hidden Weights (w_{hj}) . . . 118
58.3 Implementation Details . . . 119

59 Deep Learning Overview 119

60 Lab Work: Neural Networks for Digit Recognition 119
60.1 The MNIST Dataset . . . 120
60.2 Neural Network Structure . . . 120
60.3 Implementation Tasks . . . 121

Lecture 10: Introduction to Deep Learning and CNNs 122

61 Introduction 123

62 Recap: From Perceptrons to Multilayer Perceptrons 123
62.1 The Perceptron . . . 123
62.2 Multiclass Classification . . . 123
62.3 The Multi-layer Perceptron (MLP) . . . 124
62.4 Backpropagation . . . 124

63 Introduction to Deep Learning 125
63.1 Deep Learning Definition . . . 125
63.2 Why Now? . . . 126
63.3 Representation Learning . . . 126

64 Neural Network Modules 126
64.1 Common Modules . . . 127
64.1.1 Linear Module . . . 127
64.1.2 Sigmoid and Tanh . . . 127
64.1.3 Rectified Linear Unit (ReLU) . . . 127

65 Convolutional Neural Networks (CNNs) 128
65.1 The Convolution Operation . . . 128
65.1.1 Properties . . . 128
65.1.2 Key Concepts . . . 128
65.2 Output Dimensions . . . 129
65.3 Pooling Layers . . . 129

65.4 LeNet Architecture . . . 129
65.5 Practical Issues and Invariances . . . 129

66 Analysis of Depth **130**

67 Lab 9: Convolutional Neural Networks for Digit Recognition **130**
67.1 The MNIST Dataset . . . 130
67.2 CNN Pipeline with LeNet . . . 131
67.3 Implementation Steps . . . 131
67.4 Evaluation . . . 131
67.5 Exploration . . . 132

68 Course Summary **132**

Lecture 1: Introduction, Model Selection, and Evaluation

1 Introduction to Machine Learning

1.1 Definition

Machine Learning (ML) is a subset of Artificial Intelligence (AI) focused on developing algorithms that improve their performance on a specific task through experience (data) without being explicitly programmed for that specific instance.

- **Artificial Intelligence**: A program that can sense, reason, act, and adapt.
- **Machine Learning**: Algorithms whose performance improves as they are exposed to more data over time.
- **Deep Learning**: A subset of ML where multilayered neural networks learn from vast amounts of data.

1.2 Why Learning?

Learning is essential when:

1. Human expertise does not exist (e.g., specific bioinformatics tasks).
2. Humans cannot explain their expertise (e.g., speech recognition, computer vision).
3. The solution changes over time (e.g., routing, financial markets).
4. The solution needs to be adapted to particular cases (e.g., personalization).

1.3 Basic Machine Learning Pipeline

The standard pipeline involves:

1. **Data Collection**: Gathering raw data (images, text, measurements).
2. **Feature Extraction**: Transforming raw data into a suitable representation (feature vectors).
3. **Model Training**: Selecting a hypothesis class and optimizing parameters.
4. **Model Evaluation**: Testing the model on unseen data.

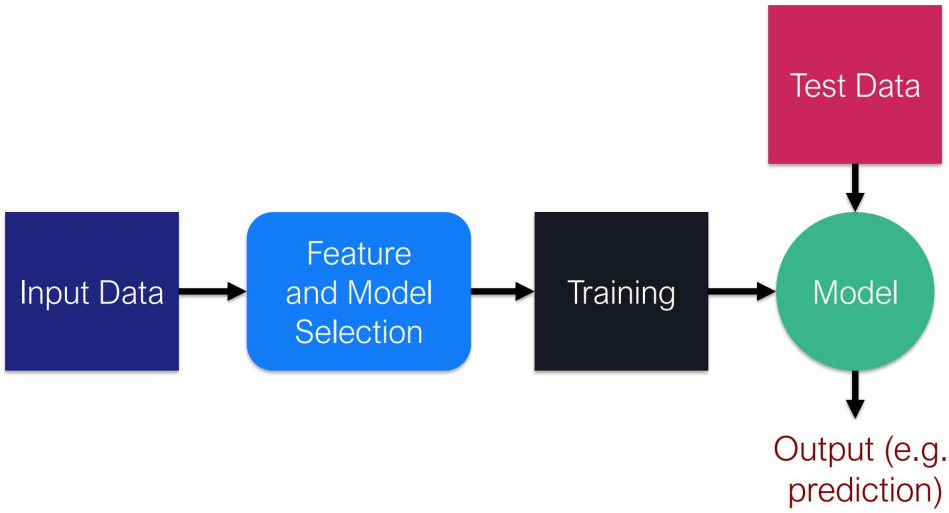


Figure 1: The Machine Learning Pipeline: From Training to Prediction.

2 Types of Machine Learning

2.1 Unsupervised Learning

The goal is to learn a new representation of the data or find hidden structures. The data $\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$ has no labels.

- **Clustering**: Grouping similar data points together (e.g., customer segmentation, topic modeling).
- **Dimensionality Reduction**: Finding a lower-dimensional representation that preserves essential information (e.g., PCA). Useful for visualization and compression.

2.2 Supervised Learning

The goal is to make predictions. We have a labeled dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$. We want to learn a mapping $f : \mathcal{X} \to \mathcal{Y}$.

- **Classification**: The output y is discrete (categorical).
  - Binary: $y \in \{0, 1\}$ or $\{-1, +1\}$.
  - Multiclass: $y \in \{0, 1, \ldots, k\}$.
- **Regression**: The output y is continuous ($y \in \mathbb{R}$).

3 Formalization of Supervised Learning

3.1 The Setup

We assume data is generated by an unknown distribution $\mathcal{D}$. We are given n independent and identically distributed (i.i.d.) samples:

$$S = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\} \sim \mathcal{D}^n$$

Our goal is to find a predictor f that minimizes the error on new, unseen data drawn from the same distribution.

3.2 Building a Model: 3 Steps

1. **Hypothesis Class ($\mathcal{F}$)**: The set of all possible functions we consider (e.g., linear functions, neural networks). We choose $f \in \mathcal{F}$.
2. **Loss Function ($\mathcal{L}$)**: A function that quantifies the error of a prediction.
$$\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$$
$$(y, \hat{y}) \mapsto \mathcal{L}(y, \hat{y})$$
3. **Optimization Algorithm**: A method to find the specific $f^* \in \mathcal{F}$ that minimizes the loss on the training data.

3.3 Loss Functions

Different tasks require different loss functions.

3.3.1 For Classification ($y \in \{-1, +1\}$)

- **0/1 Loss**: Correct classification costs 0, incorrect costs 1.
$$\mathcal{L}(y, f(x)) = \mathbb{I}(y \neq f(x)) = \begin{cases} 0 & \text{if } y = f(x) \\ 1 & \text{otherwise} \end{cases}$$
This is non-convex and hard to optimize directly.
- **Hinge Loss** (used in SVM): $\mathcal{L}(y, f(x)) = \max(0, 1 - yf(x))$.
- **Logistic Loss**: $\mathcal{L}(y, f(x)) = \log(1 + \exp(-yf(x)))$.

3.3.2 For Regression ($y \in \mathbb{R}$)

- **Squared Loss (L2)**: Penalizes large errors heavily.
$$\mathcal{L}(y, f(x)) = (y - f(x))^2$$
- **Absolute Loss (L1)**: More robust to outliers.
$$\mathcal{L}(y, f(x)) = |y - f(x)|$$

3.4 Empirical Risk Minimization (ERM)

Ideally, we want to minimize the **True Risk** (Expected Loss) over the data distribution:
$$R(f) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\mathcal{L}(y, f(x))]$$
Since we don't know $\mathcal{D}$, we cannot compute this directly. Instead, we minimize the **Empirical Risk** (Training Error):
$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y^{(i)}, f(x^{(i)}))$$
The optimal function f^* is:
$$f^* = \arg\min_{f\in\mathcal{F}} \hat{R}(f)$$

4 Optimization

4.1 Convex Optimization

Optimization is the engine of machine learning. We often prefer **convex** problems because they are easier to solve (local minima are global minima).

4.1.1 Definitions

- **Convex Set**: A set S is convex if for any $u, v \in S$ and $t \in [0, 1]$, the point $tu+(1-t)v$ is also in S (the line segment between points lies inside the set).
- **Convex Function**: A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if its domain is convex and:

$$f(tu + (1 - t)v) \leq tf(u) + (1 - t)f(v)$$

4.1.2 Characterizations of Convex Functions

If f is differentiable:

- **First-order condition**:

$$f(v) \geq f(u) + \nabla f(u)^T(v - u) \quad \forall u, v$$

  The tangent plane is always an under-estimator of the function.

- **Second-order condition** (if twice differentiable):

$$\nabla^2 f(u) \succeq 0 \quad \text{(The Hessian is Positive Semi-Definite)}$$

4.2 Gradient Descent

When a closed-form solution (like setting $\nabla f(u) = 0$) isn't possible or efficient, we use iterative methods.

Gradient Descent Algorithm:

1. Initialize $u^{(0)}$ randomly.
2. Repeat until convergence:

$$u^{(k+1)} = u^{(k)} - \alpha_k \nabla f(u^{(k)})$$

   where α_k is the **step size** or **learning rate**.

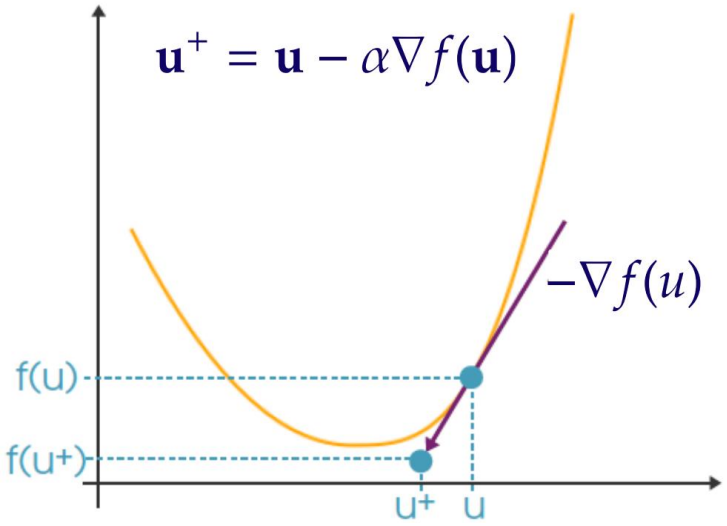


Figure 2: Gradient Descent visualization.

5 Model Selection and Generalization

5.1 Overfitting vs. Underfitting

- **Underfitting (High Bias)**: The model is too simple to capture the underlying structure of the data (e.g., fitting a line to a curve). Both training and test errors are high.
- **Overfitting (High Variance)**: The model is too complex and fits the noise in the training data (e.g., a high-degree polynomial passing through every point). Training error is low, but test error is high.

5.2 Bias-Variance Decomposition

We assume the true relationship is $y = f(x) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$. We learn a model $\hat{f}(x)$ from a standard training set. The expected squared error on a specific unseen example x can be decomposed into three terms.

Derivation: Let $\mathbb{E}$ denote the expectation over different realizations of the training set. Note that $f(x)$ is independent of the training set (deterministic), while $\hat{f}(x)$ depends on it. y is the noisy observation $f(x) + \epsilon$.

$$\begin{aligned}
\text{MSE} &= \mathbb{E}\left[(y - \hat{f}(x))^2\right] \\
&= \mathbb{E}\left[(f(x) + \epsilon - \hat{f}(x))^2\right] \\
&= \mathbb{E}\left[(f(x) - \hat{f}(x))^2 + \epsilon^2 + 2\epsilon(f(x) - \hat{f}(x))\right]
\end{aligned}$$

Since ϵ is noise with zero mean and independent of the model $\hat{f}$:

$$\mathbb{E}[\epsilon] = 0, \quad \mathbb{E}[\epsilon^2] = \sigma_\epsilon^2, \quad \mathbb{E}[\epsilon(f - \hat{f})] = 0$$

Thus:

$$\text{MSE} = \mathbb{E}\left[(f(x) - \hat{f}(x))^2\right] + \sigma_\epsilon^2$$

Now consider the term $\mathbb{E}[(f(x)-\hat{f}(x))^2]$. Let $\bar{f}(x) = \mathbb{E}[\hat{f}(x)]$ be the average prediction of our model over infinite training sets. We add and subtract $\bar{f}(x)$:

$$\begin{aligned}\mathbb{E}\left[(f(x) - \hat{f}(x))^2\right] &= \mathbb{E}\left[(f(x) - \bar{f}(x) + \bar{f}(x) - \hat{f}(x))^2\right] \\ &= \mathbb{E}\left[(f(x) - \bar{f}(x))^2\right] + \mathbb{E}\left[(\bar{f}(x) - \hat{f}(x))^2\right] + 2\mathbb{E}[(f(x) - \bar{f}(x))(\bar{f}(x) - \hat{f}(x))]\end{aligned}$$

The cross term vanishes because $f(x) - \bar{f}(x)$ is constant (non-random) and $\mathbb{E}[\bar{f}(x) - \hat{f}(x)] = \bar{f}(x) - \mathbb{E}[\hat{f}(x)] = 0$.

This leaves us with:

1. **Bias Squared**: $(f(x) - \bar{f}(x))^2$. Error due to erroneous assumptions (underfitting).
2. **Variance**: $\mathbb{E}[(\hat{f}(x) - \bar{f}(x))^2]$. Sensitivity to small fluctuations in the training set (overfitting).

Final Decomposition:

$$\text{MSE} = \text{Bias}^2(\hat{f}(x)) + \text{Var}(\hat{f}(x)) + \text{Noise}(\sigma_\epsilon^2)$$

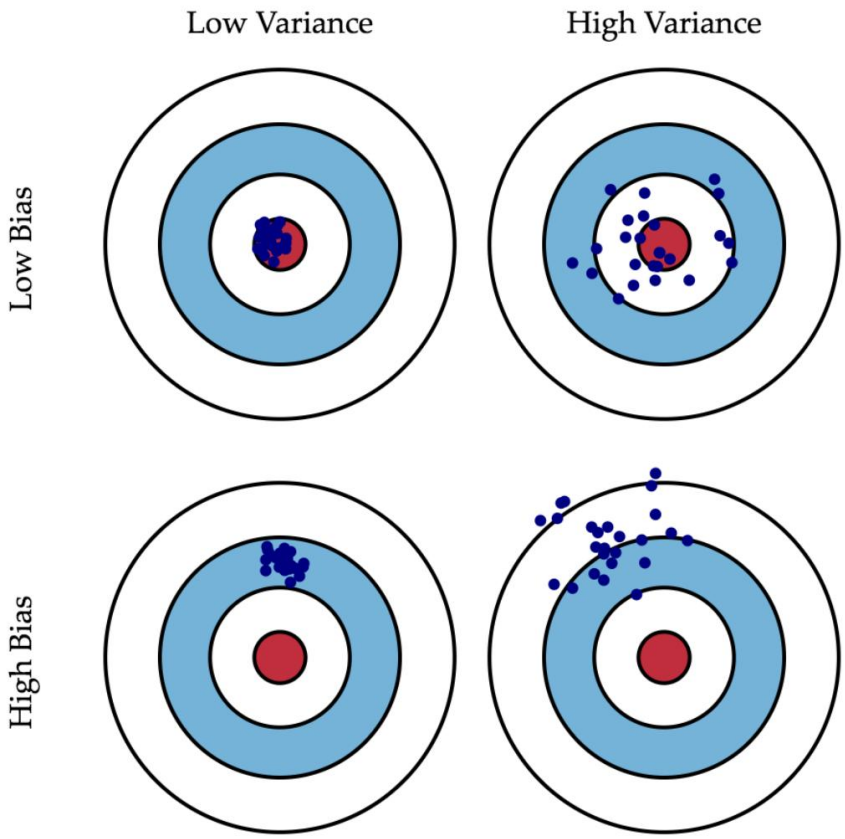


Figure 3: Visualizing Bias and Variance with targets. High Bias misses the center; High Variance scatters around.

5.3 Cross-Validation

To estimate generalization error and select hyperparameters without touching the test set, we use K**-Fold Cross-Validation**:

1. Split training data into K equal partitions (folds).
2. For $i = 1$ to K:
    - Train on all folds except i.
    - Validate (measure error) on fold i.
3. Average the K validation errors.

Common choices are $K = 5$ or $K = 10$. This provides a more robust estimate of model performance than a single train/validation split.

6 Evaluation Metrics

6.1 Classification Metrics

For binary classification, we use the **Confusion Matrix**:

	Predicted Pos (+1)	Predicted Neg (-1)
Actual Pos (+1)	**True Positive (TP)**	False Negative (FN)
Actual Neg (-1)	False Positive (FP)	**True Negative (TN)**

Table 1: Confusion Matrix

Key metrics derived from this:

- **Accuracy**: $\frac{TP+TN}{TP+TN+FP+FN}$. (Can be misleading for imbalanced datasets).
- **Sensitivity / Recall / TPR**: $\frac{TP}{TP+FN}$. Ability to find all positive samples.
- **Specificity / TNR**: $\frac{TN}{TN+FP}$. Ability to find all negative samples.
- **Precision**: $\frac{TP}{TP+FP}$. Quality of positive predictions.
- **F1-Score**: Harmonic mean of Precision and Recall.

$$F1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN}$$

6.2 ROC Curve

The **Receiver Operating Characteristic (ROC)** curve plots TPR (y-axis) vs FPR (x-axis) at various threshold settings.

- **AUC (Area Under Curve)**: Summarizes the ROC. AUC=1 is perfect, AUC=0.5 is random guessing.

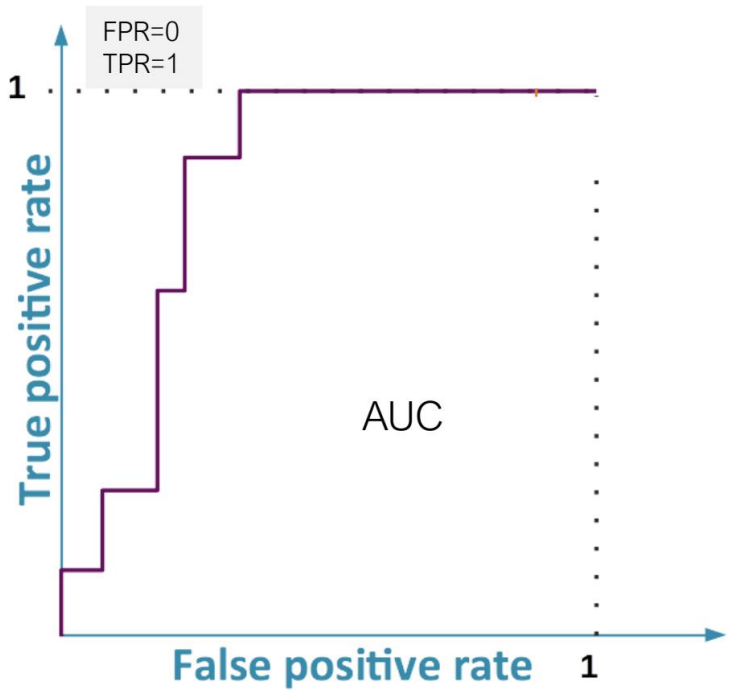


Figure 4: Example ROC Curve.

Lecture 2: Linear Regression

Contents

7 Introduction to Regression

Regression analysis is a statistical tool used to investigate the relationship between an output (dependent) variable and one or more input (independent) variables. Unlike classification, where the output is discrete (categories), in regression, the output y is continuous ($y \in \mathbb{R}$).

7.1 Goal and Applications

The goal of regression is to learn a function f that maps input features $\mathbf{x}$ to a continuous target y, i.e., $f(\mathbf{x}) \approx y$.

Examples of regression problems include:

- Predicting movie ratings (e.g., a score from 0 to 5).
- Estimating the price of a house based on its characteristics (size, location).
- Forecasting the number of social media followers.

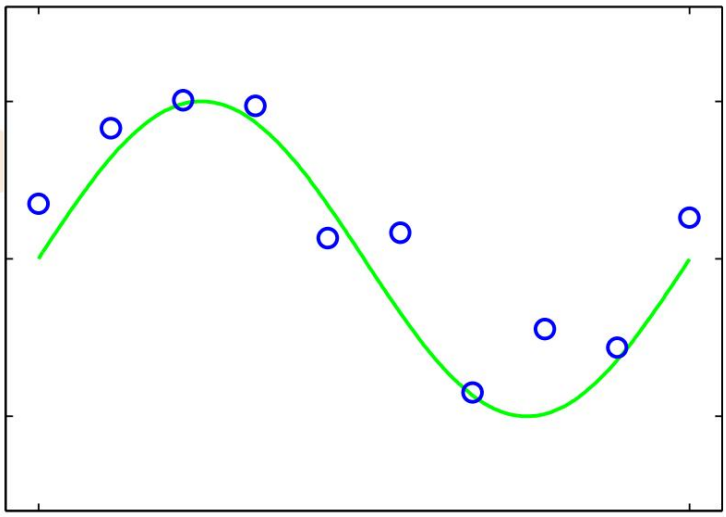


Figure 5: Example of curve fitting: The goal is to fit a curve (model) to the data points (observations).

8 Linear Regression

Linear regression is a parametric approach where we assume the relationship between the input $\mathbf{x}$ and the output y is linear.

8.1 Problem Formulation

Given a dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^K$ is the input vector of K features and $y^{(i)} \in \mathbb{R}$ is the target scalar.

The model hypothesis $h_{\boldsymbol{\theta}}(\mathbf{x})$ is defined as:

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_K x_K$$

We can define $x_0 = 1$ (the bias term input) to vectorize the notation:

$$\hat{y} = \boldsymbol{\theta}^T \mathbf{x}$$

where $\boldsymbol{\theta} = [\theta_0, \theta_1, \ldots, \theta_K]^T$ are the parameters (weights) we want to learn.

8.2 Loss Function: Least Squares

To find the best parameters $\boldsymbol{\theta}$, we need a metric to evaluate how well our model fits the data. We use the **Mean Squared Error (MSE)** or the Sum of Squared Errors. The objective function (cost function) $J(\boldsymbol{\theta})$ is defined as:

$$J(\boldsymbol{\theta}) = \frac{1}{2} \sum_{i=1}^{N} \left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{2} \sum_{i=1}^{N} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

The factor $\frac{1}{2}$ is added for mathematical convenience (it cancels out when taking the derivative).

8.3 Optimization

Our goal is to find the optimal $\boldsymbol{\theta}^*$ that minimizes the cost function:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} J(\boldsymbol{\theta})$$

There are three main approaches to solve this:

8.3.1 1. Gradient Descent (Batch GD)

Gradient descent is an iterative optimization algorithm. We start with random parameters $\boldsymbol{\theta}$ and update them by moving in the direction opposite to the gradient of the cost function.

The update rule is:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

where λ is the learning rate (step size).

Gradient Derivation: We need to compute the partial derivative of $J(\boldsymbol{\theta})$ with respect to each parameter θ_k:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_k} = \frac{\partial}{\partial \theta_k} \frac{1}{2} \sum_{i=1}^{N} (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})^2$$

Using the chain rule:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_k} = \sum_{i=1}^{N} (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}) \cdot \frac{\partial}{\partial \theta_k} (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})$$

Since $\frac{\partial}{\partial \theta_k} (\boldsymbol{\theta}^T \mathbf{x}^{(i)}) = x_k^{(i)}$:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_k} = \sum_{i=1}^{N} (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}) x_k^{(i)}$$

Thus, the gradient vector is:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{i=1}^{N} (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}) \mathbf{x}^{(i)} = \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

Pros: Guaranteed convergence to the global minimum for convex functions (like Linear Regression).
Cons: Slow for large datasets because it computes gradients over the entire dataset at every step.

8.3.2 2. Stochastic Gradient Descent (SGD)

Instead of using the entire dataset, SGD updates the parameters using the gradient of the loss for a *single* training example picked at random (or in order after shuffling).

Update rule for a single example i:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}) \mathbf{x}^{(i)}$$

This is also known as the **Least Mean Squares (LMS)** outcome.

Pros: Much faster per iteration; can escape shallow local minima (though not an issue for Linear Regression); good for online learning.
Cons: Noisy updates; convergence to the exact minimum is not guaranteed (it oscillates around it), requiring a decaying learning rate.

8.3.3 3. Closed-Form Solution (Normal Equation)

For Linear Least Squares, we can solve for $\boldsymbol{\theta}$ analytically by setting $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = 0$.

$$\mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = 0 \implies \mathbf{X}^T \mathbf{X} \boldsymbol{\theta} = \mathbf{X}^T \mathbf{y}$$
$$\boldsymbol{\theta}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

where $\mathbf{X}$ is the $N \times (K+1)$ design matrix (including bias column) and $\mathbf{y}$ is the target vector.

Pros: Exact solution in one step; no need to tune learning rate.
Cons: Computationally expensive for large K ($(\mathbf{X}^T\mathbf{X})^{-1}$ is $O(K^3)$); memory intensive.

9 Model Complexity and Generalization

Linear models might be too simple to capture complex patterns (Underfitting). We can increase complexity by using polynomial regression, i.e., adding powers of features as new features.

$$y(x, \boldsymbol{\theta}) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_M x^M$$

However, as we increase complexity (order M), the model may start to fit the noise in the training data (Overfitting).

9.1 The Bias-Variance Tradeoff

The generalization error can be decomposed into three components: Bias, Variance, and Irreducible Noise. This decomposition is crucial for understanding overfitting and underfitting.

Let our model prediction for a point $\mathbf{x}$ be $\hat{f}(\mathbf{x})$ and the true target be $y = f(\mathbf{x}) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$.

We want to minimize the expected Mean Squared Error on an unseen sample $\mathbf{x}$:

$$MSE(\mathbf{x}) = \mathbb{E}[(y - \hat{f}(\mathbf{x}))^2]$$

Definitions:

- **Bias**: $\text{Bias}[\hat{f}(\mathbf{x})] = \mathbb{E}[\hat{f}(\mathbf{x})] - f(\mathbf{x})$. Measures how far the average prediction (over different training sets) is from the truth. High bias $\implies$ Underfitting.
- **Variance**: $\text{Var}[\hat{f}(\mathbf{x})] = \mathbb{E}[(\hat{f}(\mathbf{x}) - \mathbb{E}[\hat{f}(\mathbf{x})])^2]$. Measures how much the prediction changes if we train on a different dataset. High variance $\implies$ Overfitting.
- **Noise**: σ^2. The irreducible error due to ϵ.

9.1.1 Derivation of the Decomposition

We expand the expansion of the expected MSE term:

$$\mathbb{E}[(y - \hat{f})^2] = \mathbb{E}[y^2 + \hat{f}^2 - 2y\hat{f}]$$

Recall that $\mathbb{E}[y] = f(\mathbf{x})$ and $\text{Var}[y] = \sigma^2$, so $\mathbb{E}[y^2] = \text{Var}[y] + (\mathbb{E}[y])^2 = \sigma^2 + f(\mathbf{x})^2$.

Now focus on $\mathbb{E}[(y - \hat{f})^2]$. Let's treat $\hat{f}$ as the random variable (dependent on the dataset).

$$\begin{aligned}\mathbb{E}[(y - \hat{f})^2] &= \mathbb{E}[((y - f) + (f - \hat{f}))^2] \\ &= \mathbb{E}[(y - f)^2] + \mathbb{E}[(f - \hat{f})^2] + 2\mathbb{E}[(y - f)(f - \hat{f})]\end{aligned}$$

The first term $\mathbb{E}[(y-f)^2] = \mathbb{E}[\epsilon^2] = \sigma^2$ (Noise). The cross term vanishes because $\mathbb{E}[y-f] = \mathbb{E}[\epsilon] = 0$ and ϵ is independent of $\hat{f}$ (which is fixed for a given model/training set, and we average over datasets).

$$\begin{aligned}\mathbb{E}[(y - f)(f - \hat{f})] &= \mathbb{E}_{y|\mathbf{x}}[y - f] \cdot \mathbb{E}_{\mathcal{D}}[f - \hat{f}] \\ &= 0 \cdot (\dots) = 0\end{aligned}$$

Now analyze the second term $\mathbb{E}[(f - \hat{f})^2]$:

$$\begin{aligned}\mathbb{E}[(f - \hat{f})^2] &= \mathbb{E}[(f - \mathbb{E}[\hat{f}] + \mathbb{E}[\hat{f}] - \hat{f})^2] \\ &= \mathbb{E}[(f - \mathbb{E}[\hat{f}])^2] + \mathbb{E}[(\mathbb{E}[\hat{f}] - \hat{f})^2] + 2\mathbb{E}[(f - \mathbb{E}[\hat{f}])(\mathbb{E}[\hat{f}] - \hat{f})]\end{aligned}$$

Since f and $\mathbb{E}[\hat{f}]$ are constants (not random variables), the cross term contains $\mathbb{E}[\mathbb{E}[\hat{f}] - \hat{f}] = \mathbb{E}[\hat{f}] - \mathbb{E}[\hat{f}] = 0$. The term $(f - \mathbb{E}[\hat{f}])^2$ is exactly the square of the **Bias**. The term $\mathbb{E}[(\mathbb{E}[\hat{f}] - \hat{f})^2]$ is exactly the **Variance**.

Thus, we arrive at the decomposition:

$$\boxed{MSE = \sigma^2 + \text{Bias}^2 + \text{Variance}}$$

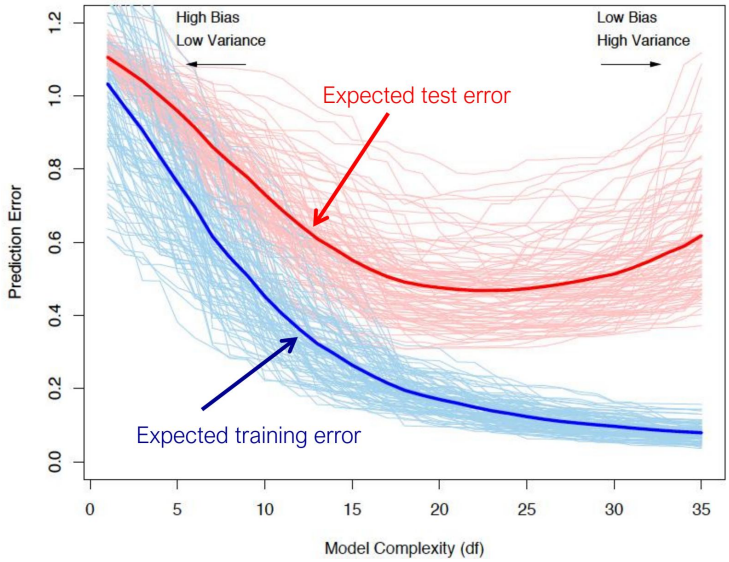


Figure 6: The tradeoff: Increasing model complexity decreases bias but increases variance. The total error is minimized at an optimal complexity level.

9.2 Regularization

Regularization adds a penalty term to the loss function to effectively reduce model complexity (reduce variance) at the cost of slight bias.

- **Ridge Regression (L2)**:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum(\hat{y}^{(i)} - y^{(i)})^2 + \lambda\|\boldsymbol{\theta}\|_2^2$$

  Shrinks parameters towards zero. Weights are small but non-zero.

- **Lasso Regression (L1)**:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum(\hat{y}^{(i)} - y^{(i)})^2 + \lambda\|\boldsymbol{\theta}\|_1$$

  Promotes **sparsity**. Can set some coefficients exactly to zero (feature selection).

- **Elastic Net**: Combination of L1 and L2.

10 Classification as Regression?

We might be tempted to use Linear Regression for a binary classification problem where $y \in \{-1, 1\}$ (or $\{0, 1\}$). The idea is to fit a linear model $y(\mathbf{x}) = \boldsymbol{\theta}^T\mathbf{x}$ and then define a decision rule based on the output.

10.1 Decision Rule

For a binary classification, we can use the sign of the prediction to assign the class:

$$\hat{y} = \begin{cases} 1 & \text{if } \boldsymbol{\theta}^T\mathbf{x} \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

This defines a **linear classifier**. The decision boundary is the hyperplane defined by $\boldsymbol{\theta}^T\mathbf{x} = 0$, which separates the input space into two half-spaces.

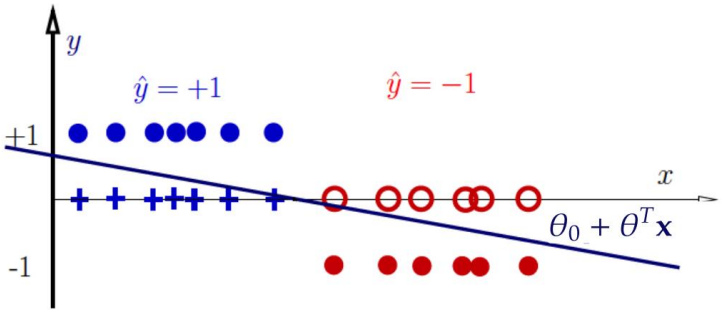


Figure 7: One-dimensional example: The classifier has the form $y = \theta_0 + \theta_1 x$. The decision boundary is the point where the line crosses zero.

The geometry of the boundary depends on the dimensions:

- **1-D**: The boundary is a point (threshold).
- **2-D**: The boundary is a line.
- **3-D**: The boundary is a plane.

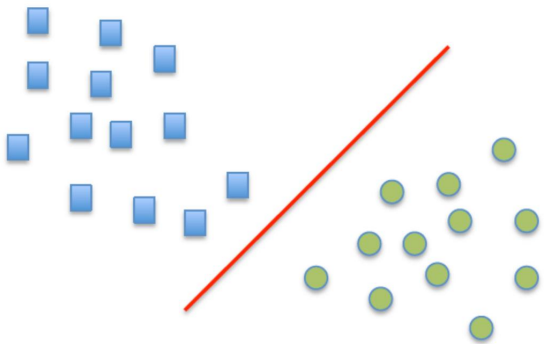


Figure 8: 2-D Decision Boundary (Line)

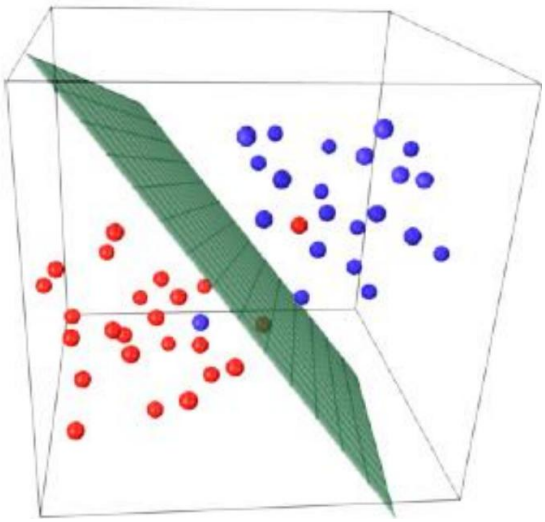


Figure 9: 3-D Decision Boundary (Plane)

10.2 Limitations

While simple, using Least Squares Linear Regression for classification has significant drawbacks:

10.2.1 1. Unbounded Predictions

The linear function $\boldsymbol{\theta}^T\mathbf{x}$ returns a continuous real number relative to the distance from the decision boundary. However, purely as a classification score, it can be arbitrarily

large or small. Interpreting these values as probabilities is impossible since they are not bounded in $[0, 1]$.

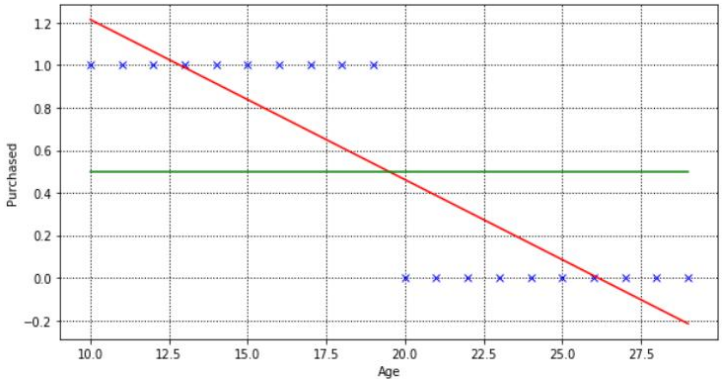


Figure 10: Linear regression predicts continuous values, which may not align well with discrete class labels.

10.2.2 2. Sensitivity to Outliers

Linear regression tries to minimize the squared error for *all* points. If we have an "outlier" (a point that is correctly classified but very far from the decision boundary), the squared error term $(\boldsymbol{\theta}^T\mathbf{x} - y)^2$ for that point becomes huge. To minimize this error, the regression line will tilt towards the outlier, potentially shifting the decision boundary and causing misclassification of normal points.

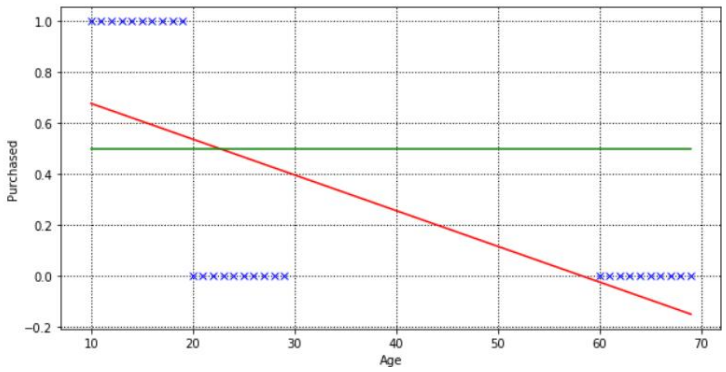


Figure 11: Sensitivity to outliers: Adding a single point far from the boundary (even if correctly classified) pulls the regression line (green vs blue), shifting the threshold and causing misclassifications near the boundary.

This sensitivity makes standard Linear Regression a poor choice for classification tasks compared to Logistic Regression, which uses a squashing function (sigmoid) to mitigate the influence of points far from the boundary.

11 Logistic Regression

Logistic Regression is better suited for binary classification. It models the probability that an input $\mathbf{x}$ belongs to the positive class ($y = 1$).

11.1 The Sigmoid Function

To squash the linear output $\boldsymbol{\theta}^T\mathbf{x}$ into the range $[0, 1]$, we use the **sigmoid (logistic)** function $\sigma(z)$:

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

So, the model hypothesis is:

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = P(y = 1|\mathbf{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^T\mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T\mathbf{x}}}$$

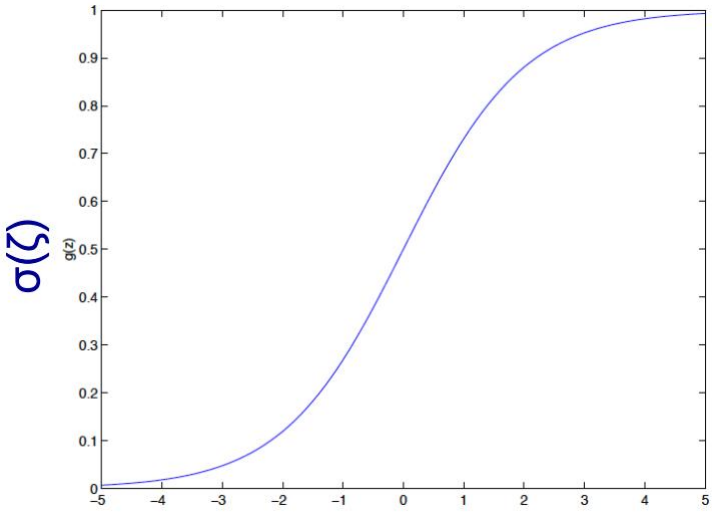


Figure 12: The Sigmoid function maps any real number to $(0, 1)$.

11.2 Decision Boundary

We classify $y = 1$ if $P(y = 1|\mathbf{x}) \geq 0.5$, which corresponds to $\boldsymbol{\theta}^T\mathbf{x} \geq 0$. The decision boundary $\boldsymbol{\theta}^T\mathbf{x} = 0$ is a linear hyperplane.

11.3 Likelihood and Cost Function

To find the optimal parameters $\boldsymbol{\theta}$ for Logistic Regression, we use the principle of **Maximum Likelihood Estimation (MLE)**. We want to find the parameters that make our observed data most probable.

11.3.1 Probabilistic Interpretation

For each individual training example $(\mathbf{x}^{(i)}, y^{(i)})$, our model predicts a single probability $h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})$. Since y is binary (0 or 1), we can model it using a **Bernoulli distribution**:

- $P(y = 1|\mathbf{x}; \boldsymbol{\theta}) = h_{\boldsymbol{\theta}}(\mathbf{x})$
- $P(y = 0|\mathbf{x}; \boldsymbol{\theta}) = 1 - h_{\boldsymbol{\theta}}(\mathbf{x})$

We can combine these two cases into a single mathematical expression for the probability of a given label $y^{(i)}$:

$$P(y^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\theta}) = (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{y^{(i)}}(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{1-y^{(i)}}$$

Notice how the exponents $y^{(i)}$ and $1 - y^{(i)}$ act as "switches": if $y^{(i)} = 1$, the second term becomes $(\dots)^0 = 1$; if $y^{(i)} = 0$, the first term becomes $(\dots)^0 = 1$.

11.3.2 The Likelihood Function

Assuming that all N training examples are independent and identically distributed (i.i.d.), the probability of the entire dataset (the *likelihood*) is the product of individual probabilities:

$$L(\boldsymbol{\theta}) = P(\mathbf{y}|\mathbf{X};\boldsymbol{\theta}) = \prod_{i=1}^{N} P(y^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\theta}) = \prod_{i=1}^{N}(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{y^{(i)}}(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{1-y^{(i)}}$$

11.3.3 Log-Likelihood

Maximizing a product of many small probabilities is computationally difficult (due to numerical underflow) and mathematically tedious. To simplify, we take the natural logarithm, converting the product into a sum. Since the log is a monotonically increasing function, the $\boldsymbol{\theta}$ that maximizes $L(\boldsymbol{\theta})$ also maximizes $\ln L(\boldsymbol{\theta})$.

$$\mathcal{L}(\boldsymbol{\theta}) = \ln L(\boldsymbol{\theta}) = \sum_{i=1}^{N} \ln\left[(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{y^{(i)}}(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{1-y^{(i)}}\right]$$

Using log properties ($\ln A^B = B \ln A$):

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \left[y^{(i)} \ln(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \ln(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))\right]$$

11.3.4 The Cost Function: Binary Cross-Entropy

In Machine Learning, we usually prefer to *minimize* a cost rather than *maximize* a likelihood. We therefore define our cost function $J(\boldsymbol{\theta})$ as the **Negative Log-Likelihood** averaged over the dataset:

$$\boxed{J(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N} \left[y^{(i)} \ln(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \ln(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))\right]}$$

This is known as the **Log-Loss** or **Binary Cross-Entropy**. It penalizes the model heavily when it predicts a probability far from the actual label (e.g., predicting 0.01 when $y = 1$).

11.4 Gradient Descent for Logistic Regression

We update $\boldsymbol{\theta}$ to minimize $J(\boldsymbol{\theta})$. The gradient of the Negative Log-Likelihood turns out to be look identical to Linear Regression's (but with a different hypothesis $\hat{y}$):

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\mathbf{X}^T(\hat{\mathbf{y}} - \mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N}(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)})\mathbf{x}^{(i)}$$

12 Lab 1: Ridge and Logistic Regression

This section summarizes the theoretical and practical components of Lab 1, which explores regularization in linear models and the implementation of Logistic Regression.

12.1 Ridge Regression Theory

Ridge regression adds an L_2 penalty to the standard linear regression objective to prevent overfitting.

12.1.1 1. Problem Formulation

Given a dataset $\{y_i, \mathbf{x}_i\}_{i=1}^m$, the Ridge Regression objective function $J(\boldsymbol{\theta})$ is:

$$J(\boldsymbol{\theta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2$$

where $\lambda > 0$ is the regularization parameter.

12.1.2 2. Closed-Form Solution

To find the optimal $\boldsymbol{\theta}$, we compute the gradient and set it to zero:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}}[(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) + \lambda\boldsymbol{\theta}^T\boldsymbol{\theta}]$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) + 2\lambda\boldsymbol{\theta} = 2(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y} + \lambda\boldsymbol{\theta})$$

Setting $2(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} + \lambda\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y}) = 0$:

$$(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\theta} = \mathbf{X}^T\mathbf{y} \implies \boxed{\boldsymbol{\theta}^* = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}}$$

12.1.3 3. Hessian Matrix

The Hessian matrix $\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is the matrix of second-order partial derivatives:

$$\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{\partial}{\partial\boldsymbol{\theta}}[2(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y} + \lambda\boldsymbol{\theta})] = \boxed{2(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})}$$

Since $(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})$ is positive definite for $\lambda > 0$, the objective function is strictly convex, ensuring a unique global minimum.

12.2 Python Implementation: Logistic Regression

The lab implementation focuses on classifying student admissions based on two exam scores.

12.2.1 1. Core Functions

The following functions are implemented to build the classifier:

- **Sigmoid**: $g(z) = \frac{1}{1+e^{-z}}$.
- **Cost Function**: $J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^m [y^{(i)}\log(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) + (1 - y^{(i)})\log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))]$.
- **Gradient**: $\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^m (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)}$.

12.2.2 2. Optimization Pipeline

Instead of manual gradient descent, the lab uses `scipy.optimize.minimize` for efficiency:

1. **Data Loading**: Load exam scores and labels.
2. **Initialization**: Initialize $\boldsymbol{\theta}$ with zeros.
3. **Optimization**: Use the `TNC` (Truncated Newton) method to find $\boldsymbol{\theta}^*$ that minimizes the cost.
4. **Prediction**: Threshold the sigmoid output at 0.5 to assign classes.

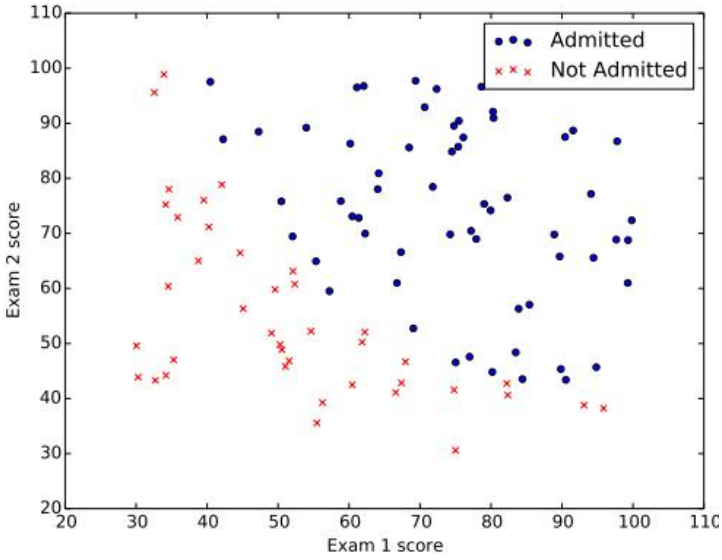


Figure 13: Lab data visualization: Exam scores plotted colored by admission status.

Note on Closed-Form Solutions: It is important to distinguish between regression models. While **Linear Regression** and **Ridge Regression** (as seen in Lab 1) have closed-form analytical solutions (like the Normal Equation), **Logistic Regression** does not. This is because the derivative of the Logistic cost function with respect to $\boldsymbol{\theta}$ involves the non-linear sigmoid function, leading to a system of transcendental equations that cannot be solved explicitly. Therefore, we must rely on iterative optimization methods like Gradient Descent or Newton's Method.

Lecture 3: Linear Classification

Contents

13 Introduction and Recap

13.1 Linear Methods Recap

In previous lectures, we explored **Linear Regression**, where the goal is to learn parameters $\boldsymbol{\theta}$ that minimize a cost function $J(\boldsymbol{\theta})$. For standard Least Squares, this is:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{2} \sum_{i=1}^{N} (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})^2$$

This can be solved either via **Gradient Descent** or the closed-form normal equation:

$$\boldsymbol{\theta} = (X^T X)^{-1} X^T \mathbf{y}$$

To prevent overfitting, we introduced regularization:

- **Ridge Regression (L_2 regularization)**: Adds $\lambda \|\boldsymbol{\theta}\|_2^2$ to the cost function.
- **LASSO (L_1 regularization)**: Adds $\lambda \|\boldsymbol{\theta}\|_1$ to the cost function, promoting sparsity.

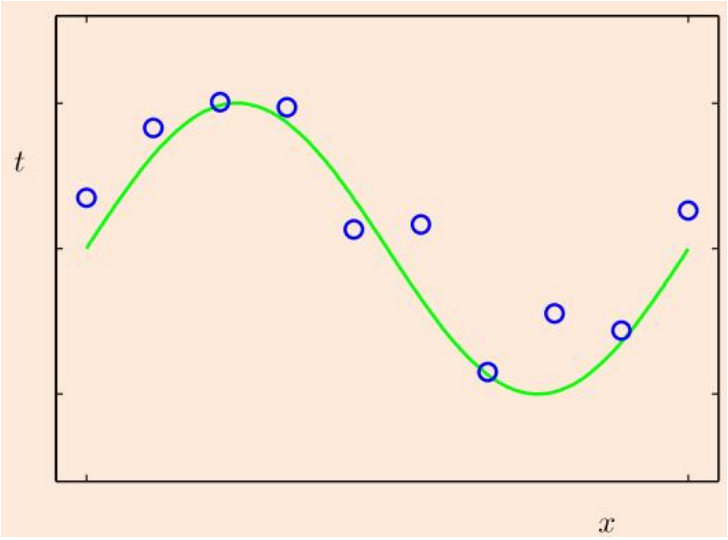


Figure 14: Linear Regression fitting a line to data points minimizing the sum of squared errors.

14 Logistic Regression

14.1 From Regression to Classification

In binary classification, we want to predict a label $y \in \{0, 1\}$ given input $\mathbf{x}$. Using a linear regression model directly is problematic because $\boldsymbol{\theta}^T \mathbf{x}$ can span from $-\infty$ to $+\infty$, whereas probabilities must lie in $[0, 1]$.

Logistic Regression models the probability that a sample belongs to class 1:

$$p(y = 1 | \mathbf{x}; \boldsymbol{\theta})$$

Instead of a hard threshold function, we map the linear output $\boldsymbol{\theta}^T\mathbf{x}$ to the $(0, 1)$ range using the **sigmoid function** $\sigma(z)$:

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

Thus, our hypothesis becomes:

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = p(y = 1|\mathbf{x}) = \sigma(\boldsymbol{\theta}^T\mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T\mathbf{x}}}$$

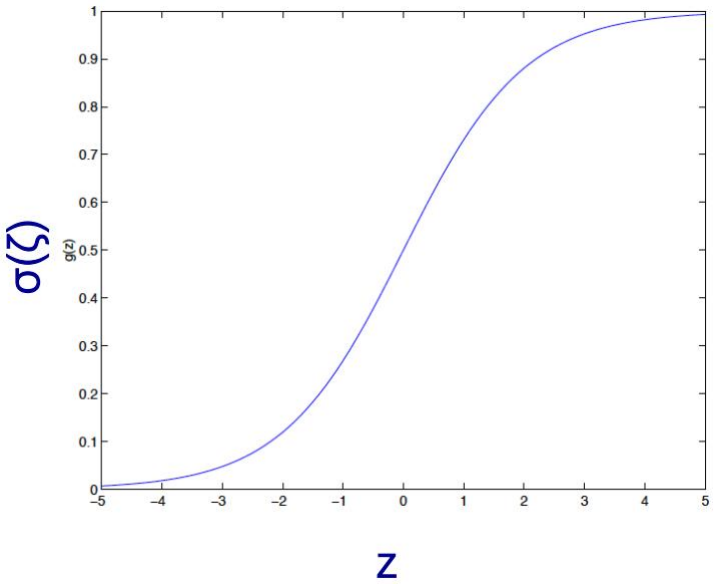


Figure 15: The Sigmoid Function $\sigma(z)$ maps any real number to the range $(0, 1)$.

The probability for the negative class is simply:

$$p(y = 0|\mathbf{x}) = 1 - p(y = 1|\mathbf{x}) = 1 - \sigma(\boldsymbol{\theta}^T\mathbf{x})$$

14.2 Probabilistic Interpretation

We assume the target variable y follows a **Bernoulli distribution** conditioned on $\mathbf{x}$:

$$p(y|\mathbf{x}; \boldsymbol{\theta}) = (h_{\boldsymbol{\theta}}(\mathbf{x}))^y(1 - h_{\boldsymbol{\theta}}(\mathbf{x}))^{1-y}$$

Notice that if $y = 1$, this simplifies to $h_{\boldsymbol{\theta}}(\mathbf{x})$, and if $y = 0$, it simplifies to $1 - h_{\boldsymbol{\theta}}(\mathbf{x})$.

14.3 Maximum Likelihood Estimation (MLE)

To train the model, we want to find $\boldsymbol{\theta}$ that maximizes the likelihood of the observed training data. Assuming m independent and identically distributed (i.i.d.) training samples, the likelihood function $L(\boldsymbol{\theta})$ is:

$$L(\boldsymbol{\theta}) = \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}; \boldsymbol{\theta}) = \prod_{i=1}^{m} (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{y^{(i)}}(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))^{1-y^{(i)}}$$

It is mathematically easier to maximize the **Log-Likelihood** $\mathcal{L}(\boldsymbol{\theta})$:

$$\mathcal{L}(\boldsymbol{\theta}) = \log L(\boldsymbol{\theta}) = \sum_{i=1}^{m} \left[y^{(i)} \log(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) \right]$$

This function is concave, meaning it has a unique global maximum.

14.3.1 Gradient Ascent

To maximize $\mathcal{L}(\boldsymbol{\theta})$, we can use **Gradient Ascent**. The update rule is:

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

The gradient of the log-likelihood with respect to a single parameter θ_j is given by (derivation omitted, but result is standard):

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \theta_j} = \sum_{i=1}^{m} (y^{(i)} - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) x_j^{(i)}$$

So the update rule in vector form is:

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \alpha \sum_{i=1}^{m} (y^{(i)} - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) \mathbf{x}^{(i)}$$

Alternatively, we can minimize the **Negative Log-Likelihood** (often called Cross-Entropy Loss) using Gradient Descent.

14.4 Decision Boundary

Once trained, we classify a new sample $\mathbf{x}_{\text{new}}$ by checking if $p(y = 1|\mathbf{x}_{\text{new}}) \geq 0.5$. Since $\sigma(z) \geq 0.5$ when $z \geq 0$, this corresponds to:

$$\boldsymbol{\theta}^T \mathbf{x}_{\text{new}} \geq 0 \implies \hat{y} = 1$$

For multiclass problems, Logistic Regression can be extended using the **Softmax** function (Multinomial Logistic Regression) or One-vs-All strategy.

15 Discriminative vs. Generative Models

Classification models can be broadly categorized into two types:

1. **Discriminative Models** (e.g., Logistic Regression, SVMs):
   - Learn $p(y|\mathbf{x})$ directly.
   - Try to find a decision boundary that separates classes.
   - Make fewer assumptions about the data distribution.
   - Generally robust to modeling errors and perform well with large datasets.

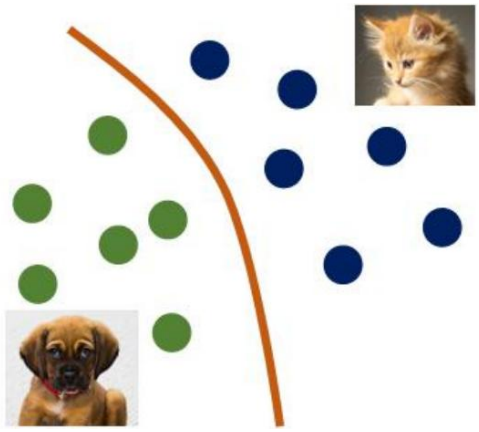


Figure 16: Discriminative approach focuses on the boundary.

2. **Generative Models** (e.g., Naïve Bayes, GDA):
   - Model how the data is generated: $p(\mathbf{x}|y)$ (Class-Conditional Density) and $p(y)$ (Class Prior).
   - Use **Bayes' Rule** to compute the posterior $p(y|\mathbf{x})$:

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})}$$

   - Can be more data-efficient (require fewer samples) if the modeling assumptions are correct.
   - Can handle missing data and outliers better in some contexts.

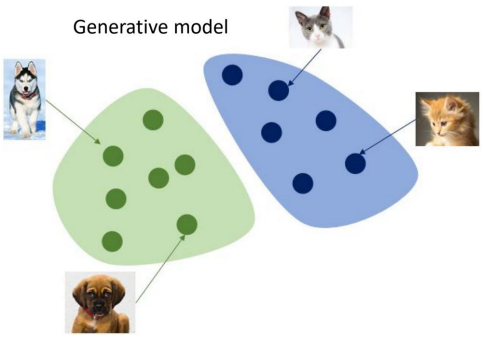


Figure 17: Generative approach models the distribution of each class.

16 Gaussian Discriminant Analysis (GDA)

GDA is a generative model where we assume that $p(\mathbf{x}|y)$ follows a multivariate normal (Gaussian) distribution.

16.1 The Model

For a binary classification problem ($y \in \{0, 1\}$), we assume:

$$\begin{aligned} p(y) &\sim \text{Bernoulli}(\phi) \\ p(\mathbf{x}|y=0) &\sim \mathcal{N}(\boldsymbol{\mu}_0, \Sigma) \\ p(\mathbf{x}|y=1) &\sim \mathcal{N}(\boldsymbol{\mu}_1, \Sigma) \end{aligned}$$

Note a key assumption in standard Linear Discriminant Analysis (LDA): both classes share the same covariance matrix Σ. If we allow different covariance matrices Σ_0 and Σ_1, fit leads to Quadratic Discriminant Analysis (QDA).

The multivariate Gaussian density is given by:

$$p(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right)$$

16.2 Parameter Estimation (MLE)

We estimate the parameters $\phi, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \Sigma$ from the data using MLE:

$$\begin{aligned} \hat{\phi} &= \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\{y^{(i)} = 1\} \quad \text{(Fraction of class 1)} \\ \hat{\boldsymbol{\mu}}_0 &= \frac{\sum_{i=1}^{N} \mathbf{1}\{y^{(i)} = 0\} \mathbf{x}^{(i)}}{\sum_{i=1}^{N} \mathbf{1}\{y^{(i)} = 0\}} \quad \text{(Mean of class 0 samples)} \\ \hat{\boldsymbol{\mu}}_1 &= \frac{\sum_{i=1}^{N} \mathbf{1}\{y^{(i)} = 1\} \mathbf{x}^{(i)}}{\sum_{i=1}^{N} \mathbf{1}\{y^{(i)} = 1\}} \quad \text{(Mean of class 1 samples)} \\ \hat{\Sigma} &= \frac{1}{N} \sum_{i=1}^{N} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_{y^{(i)}})(\mathbf{x}^{(i)} - \boldsymbol{\mu}_{y^{(i)}})^T \end{aligned}$$

16.3 Classification

To classify a new sample $\mathbf{x}$, we calculate the posterior probability using Bayes' rule. For the shared covariance case ($\Sigma_0 = \Sigma_1 = \Sigma$), the decision boundary is linear. It can be shown that the log-odds ratio is linear in $\mathbf{x}$:

$$\log \frac{p(y=1|\mathbf{x})}{p(y=0|\mathbf{x})} = \log \frac{p(\mathbf{x}|y=1)p(y=1)}{p(\mathbf{x}|y=0)p(y=0)}$$

Expanding the Gaussian terms, the quadratic parts $\mathbf{x}^T \Sigma^{-1} \mathbf{x}$ cancel out, leaving a linear function.

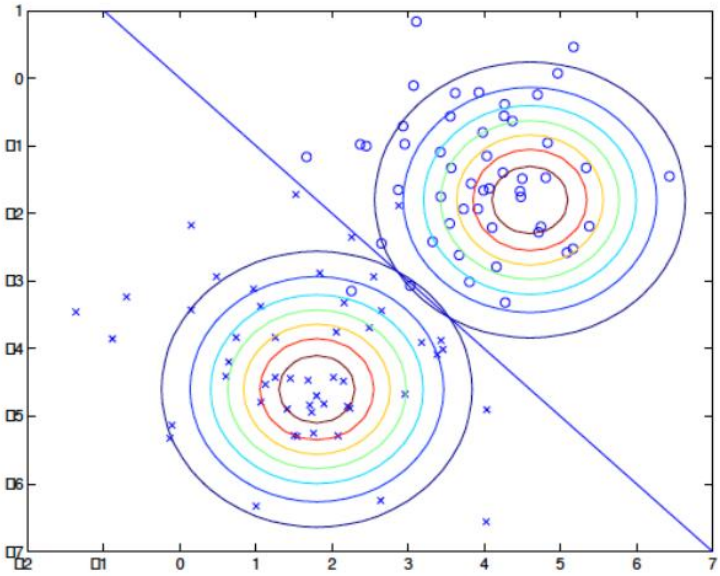


Figure 18: Gaussian Discriminant Analysis contour plots. The straight line represents the decision boundary where $p(y = 1|\mathbf{x}) = 0.5$.

17 Naïve Bayes

17.1 The Naïve Assumption

When the feature dimension d is high (e.g., text classification with vocabulary size 50,000), estimating a full covariance matrix for GDA is impossible (requires $\approx d^2/2$ parameters) without massive data.

Naïve Bayes makes a strong simplifying assumption: **The features $x_1, \ldots, x_d$ are conditionally independent given the class y.**

Mathematically:

$$p(\mathbf{x}|y) = p(x_1, \ldots, x_d|y) = \prod_{j=1}^{d} p(x_j|y)$$

This reduces particular joint distribution estimation to estimating d univariate distributions.

17.2 The Classifier

Using the independence assumption in Bayes' rule:

$$p(y = k|\mathbf{x}) \propto p(y = k) \prod_{j=1}^{d} p(x_j|y = k)$$

The predicted class $\hat{y}$ is:

$$\hat{y} = \underset{k \in \{0,\ldots,K\}}{\operatorname{argmax}} \left(p(y = k) \prod_{j=1}^{d} p(x_j|y = k) \right)$$

17.3 Example: Spam Classification

Consider email classification using a "Bag of Words" model. The feature vector $\mathbf{x} \in \{0,1\}^d$ represents the presence (1) or absence (0) of each word in the vocabulary.

17.3.1 Parameter Estimation

- $p(y = 1)$ (Prior for Spam): Fraction of spam emails in training.
- $p(x_j = 1|y = 1)$ (Likelihood): Probability that word j appears in a spam email.

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{N} \mathbf{1}\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{N} \mathbf{1}\{y^{(i)} = 1\}}$$

17.3.2 Laplace Smoothing

If a word j never appears in class k during training, the MLE estimate for probability is 0. This is problematic because it wipes out the entire product:

$$\prod_j p(x_j|y) = 0$$

To fix this, we use **Laplace Smoothing** (adding pseudo-counts):

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{N} \mathbf{1}\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\} + 1}{\sum_{i=1}^{N} \mathbf{1}\{y^{(i)} = 1\} + 2}$$

Here, we add 1 to the numerator and 2 (number of possible values for binary feature) to the denominator.

Figure 19: Spam classification example illustrating the Naïve Bayes concept.

18 Fisher Linear Discriminant Analysis (FDA)

FDA (also called LDA in dimensionality reduction context) is a method to project data onto a lower-dimensional space (e.g., a line) such that classes are best separated.

18.1 The Goal

We want to find a projection vector $\mathbf{w}$ such that when data $\mathbf{x}$ is projected onto it ($y = \mathbf{w}^T\mathbf{x}$):

1. The means of the two classes are far apart (Maximize Between-Class Variance).
2. The spread within each class is small (Minimize Within-Class Variance).

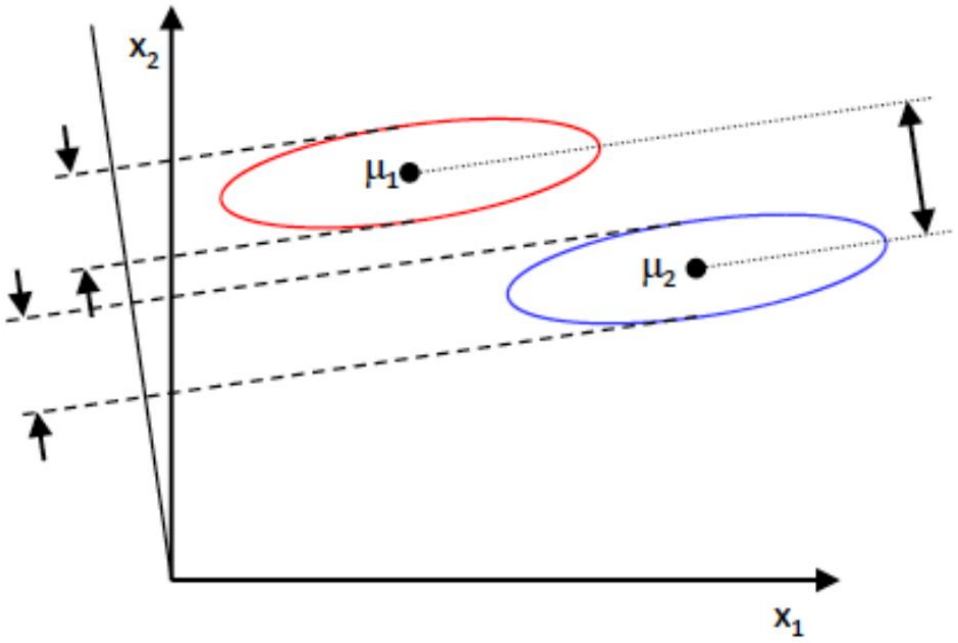


Figure 20: Left: Bad projection (classes overlap). Right: Good Fisher projection (classes are separated).

18.2 Mathematical Formulation

We want to project our data $\mathbf{x}$ onto a direction $\mathbf{w}$ to get a scalar $y = \mathbf{w}^T\mathbf{x}$. Let's analyze how the means and variances change in this projected space.

18.2.1 Projected Means

The mean of the projected data for class k is:

$$\begin{aligned}\tilde{\mu}_k &= \frac{1}{N_k}\sum_{y\in\mathcal{C}_k} y \\ &= \frac{1}{N_k}\sum_{\mathbf{x}\in\mathcal{C}_k} \mathbf{w}^T\mathbf{x} \\ &= \mathbf{w}^T\left(\frac{1}{N_k}\sum_{\mathbf{x}\in\mathcal{C}_k}\mathbf{x}\right) \\ &= \mathbf{w}^T\boldsymbol{\mu}_k\end{aligned}$$

where $\mathcal{C}_k$ denotes the set of points in Class k. We want to maximize the distance between the projected means:

$$(\tilde{\mu}_1 - \tilde{\mu}_2)^2 = (\mathbf{w}^T\boldsymbol{\mu}_1 - \mathbf{w}^T\boldsymbol{\mu}_2)^2 = (\mathbf{w}^T(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2))^2$$

This can be rewritten in matrix form:

$$(\mathbf{w}^T(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2))^2 = \mathbf{w}^T(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T\mathbf{w} = \mathbf{w}^T S_B \mathbf{w}$$

where $S_B = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T$ is the **Between-class Scatter Matrix**.

18.2.2 Projected Variances

We also want to minimize the variance (spread) within each class after projection. The scatter (unscaled variance) for class k in the projected space is:

$$\tilde{s}_k^2 = \sum_{y \in \mathcal{C}_k} (y - \tilde{\mu}_k)^2$$

Substituting $y = \mathbf{w}^T\mathbf{x}$ and $\tilde{\mu}_k = \mathbf{w}^T\boldsymbol{\mu}_k$:

$$\begin{aligned}
\tilde{s}_k^2 &= \sum_{\mathbf{x} \in \mathcal{C}_k} (\mathbf{w}^T\mathbf{x} - \mathbf{w}^T\boldsymbol{\mu}_k)^2 \\
&= \sum_{\mathbf{x} \in \mathcal{C}_k} (\mathbf{w}^T(\mathbf{x} - \boldsymbol{\mu}_k))^2 \\
&= \sum_{\mathbf{x} \in \mathcal{C}_k} \mathbf{w}^T(\mathbf{x} - \boldsymbol{\mu}_k)(\mathbf{x} - \boldsymbol{\mu}_k)^T\mathbf{w} \\
&= \mathbf{w}^T \left(\sum_{\mathbf{x} \in \mathcal{C}_k} (\mathbf{x} - \boldsymbol{\mu}_k)(\mathbf{x} - \boldsymbol{\mu}_k)^T \right) \mathbf{w} \\
&= \mathbf{w}^T S_k \mathbf{w}
\end{aligned}$$

where S_k is the within-class scatter matrix for class k. The total within-class scatter is the sum of scatters for both classes:

$$\tilde{s}_1^2 + \tilde{s}_2^2 = \mathbf{w}^T S_1 \mathbf{w} + \mathbf{w}^T S_2 \mathbf{w} = \mathbf{w}^T (S_1 + S_2)\mathbf{w} = \mathbf{w}^T S_W \mathbf{w}$$

where $S_W = S_1 + S_2$ is the **Within-class Scatter Matrix**.

18.3 Fisher's Criterion

We define the Fisher Criterion $J(\mathbf{w})$ as the ratio of between-class variance to within-class variance in the projected space:

$$J(\mathbf{w}) = \frac{(\tilde{\mu}_1 - \tilde{\mu}_2)^2}{\tilde{s}_1^2 + \tilde{s}_2^2} = \frac{\mathbf{w}^T S_B \mathbf{w}}{\mathbf{w}^T S_W \mathbf{w}}$$

18.3.1 Optimization Derivation

To find the optimal $\mathbf{w}$, we differentiate $J(\mathbf{w})$ with respect to $\mathbf{w}$ and set it to 0. We use the quotient rule for derivatives:

$$\nabla_{\mathbf{w}} \left(\frac{f(\mathbf{w})}{g(\mathbf{w})} \right) = \frac{g(\mathbf{w}) \nabla f(\mathbf{w}) - f(\mathbf{w}) \nabla g(\mathbf{w})}{g(\mathbf{w})^2}$$

Here, $f(\mathbf{w}) = \mathbf{w}^T S_B \mathbf{w}$ and $g(\mathbf{w}) = \mathbf{w}^T S_W \mathbf{w}$. Recall that $\nabla_{\mathbf{w}}(\mathbf{w}^T A \mathbf{w}) = 2A\mathbf{w}$ for a symmetric matrix A. Thus:

$$\nabla f(\mathbf{w}) = 2S_B \mathbf{w}, \quad \nabla g(\mathbf{w}) = 2S_W \mathbf{w}$$

Setting the gradient to zero:

$$\begin{aligned} \frac{(\mathbf{w}^T S_W \mathbf{w})(2S_B \mathbf{w}) - (\mathbf{w}^T S_B \mathbf{w})(2S_W \mathbf{w})}{(\mathbf{w}^T S_W \mathbf{w})^2} &= 0 \\ (\mathbf{w}^T S_W \mathbf{w}) S_B \mathbf{w} - (\mathbf{w}^T S_B \mathbf{w}) S_W \mathbf{w} &= 0 \\ (\mathbf{w}^T S_W \mathbf{w}) S_B \mathbf{w} &= (\mathbf{w}^T S_B \mathbf{w}) S_W \mathbf{w} \end{aligned}$$

Dividing both sides by the scalar $(\mathbf{w}^T S_W \mathbf{w})$:

$$S_B \mathbf{w} = \left(\frac{\mathbf{w}^T S_B \mathbf{w}}{\mathbf{w}^T S_W \mathbf{w}} \right) S_W \mathbf{w}$$

Using definitions $J(\mathbf{w}) = \lambda$, we get the **Generalized Eigenvalue Problem**:

$$S_B \mathbf{w} = \lambda S_W \mathbf{w}$$

If S_W is invertible, we can write:

$$S_W^{-1} S_B \mathbf{w} = \lambda \mathbf{w}$$

18.3.2 Solving for w

For a 2-class problem, notice that $S_B \mathbf{w}$ is a vector in the direction of $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$.

$$\begin{aligned} S_B \mathbf{w} &= (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T \mathbf{w} \\ &= (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \underbrace{[(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T \mathbf{w}]}_{\text{scalar } c} \\ &= c(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \end{aligned}$$

So the equation $S_B \mathbf{w} = \lambda S_W \mathbf{w}$ becomes:

$$c(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) = \lambda S_W \mathbf{w}$$

Since we only care about the direction of $\mathbf{w}$ (not the magnitude), we can ignore the scalar constants c and λ:

$$\mathbf{w}^* \propto S_W^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$$

This gives us the optimal projection direction.

18.4 Algorithm Summary

Step-by-step for Binary FDA:

1. Compute class means $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2$.
2. Compute within-class scatter matrix S_W.
3. Compute optimal projection $\mathbf{w} = S_W^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$.
4. Project data onto $\mathbf{w}$ to get a 1D scalar for classification or visualization.

19 Lab 2: Discriminant Analysis

19.1 Naive Bayes Exercise

In the lab, we considered a binary classification problem with two classes C_1, C_2 and a feature vector $\mathbf{x} = (X_1, X_2)$. Given class-conditional probabilities $P(X_j|C_i)$ and equal priors $P(C_1) = P(C_2) = 0.5$, we calculated the posterior probabilities for a sample $\mathbf{x} = (-1, 1)$:

$$
\begin{aligned}
P(C_1|\mathbf{x}) &= \frac{P(\mathbf{x}|C_1)P(C_1)}{P(\mathbf{x})} \\
P(\mathbf{x}|C_1) &= P(X_1 = -1|C_1)P(X_2 = 1|C_1) = 0.2 \times 0.1 = 0.02 \\
P(\mathbf{x}|C_2) &= P(X_1 = -1|C_2)P(X_2 = 1|C_2) = 0.3 \times 0.6 = 0.18 \\
P(C_1|\mathbf{x}) &= \frac{0.02 \times 0.5}{(0.02 \times 0.5) + (0.18 \times 0.5)} = \frac{0.01}{0.1} = 0.1
\end{aligned}
$$

Thus, predicting Class 2 with $P(C_2|\mathbf{x}) = 0.9$.

19.2 FDA Implementation

We implemented the Fisher Discriminant Analysis algorithm to solve a classification problem on the **Wine Dataset** (178 samples, 13 features, 3 isotropic classes).

19.2.1 Algorithm Steps

1. **Compute Scatter Matrices**:
   - Within-class scatter $S_W = \sum_{j=1}^{K} \sum_{\mathbf{x} \in C_j} (\mathbf{x} - \mathbf{m}_j)(\mathbf{x} - \mathbf{m}_j)^T$
   - Between-class scatter $S_B = \sum_{j=1}^{K} n_j (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^T$
2. **Solve Eigenvalue Problem**: Find eigenvectors of $S_W^{-1} S_B$ corresponding to the $K - 1$ largest eigenvalues.
3. **Project Data**: Project samples $\mathbf{X}$ onto the space defined by these eigenvectors to obtain $\mathbf{X}_{\text{FDA}}$.
4. **Classification**: Project class centroids $\mathbf{m}_j$ to $\mathbf{m}_j^{\text{FDA}}$. Assign new samples to the class of the nearest projected centroid.

19.2.2 Python Pipeline

The implementation involved:

1. **Data Loading**: Reading `wine_data.csv` and splitting into training (100 samples) and test (78 samples) sets.

2. **Training**: Calling `my_LDA(trainingData, trainingLabels)` to compute the projection matrix W and projected centroids.

3. **Prediction**: Using `predict()` to classify test samples based on Euclidean distance in the projected space.

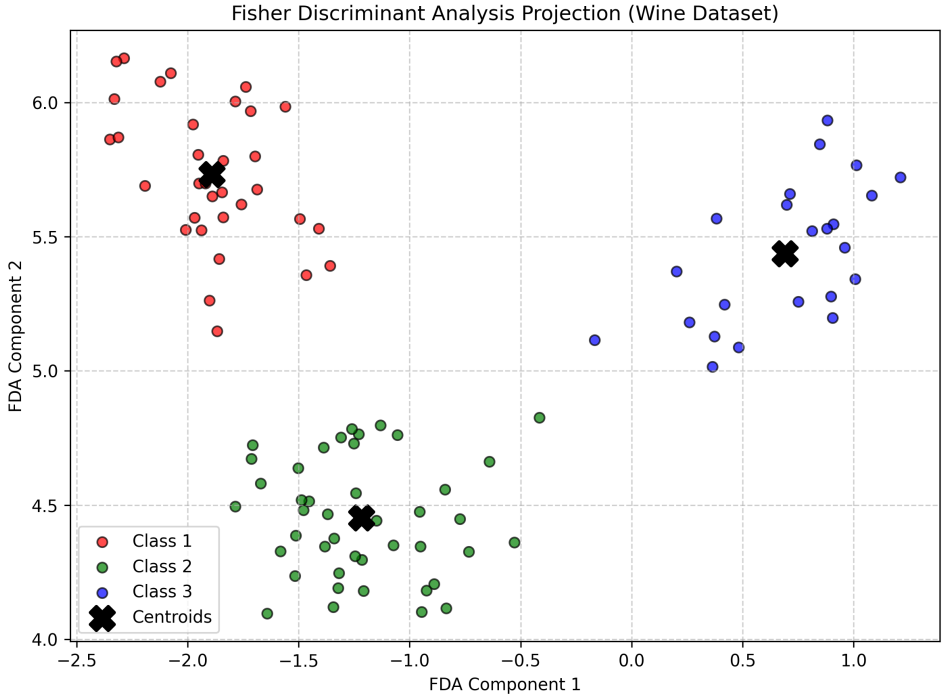


Figure 21: FDA projection of the Wine dataset onto a 2D space. The classes are well-separated, illustrating the effectiveness of the method.

Lecture 4: Support Vector Machines

20 Introduction and Recap

Support Vector Machines (SVMs) are a powerful class of supervised learning algorithms used for classification and regression (- though primarily classification). Before diving into SVMs, it is useful to briefly recap the supervised learning models tailored for classification and how they relate to what we will cover.

20.1 Supervised Learning Review

In supervised learning, we are often interested in finding a model that minimizes a specific objective function.

- **Linear Regression** (Least Squares): Finds parameters $\boldsymbol{\theta}$ that minimize the sum of squared errors:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{N}(\boldsymbol{\theta}^T\mathbf{x}^{(i)} - y^{(i)})^2$$

  This has a closed-form solution $\boldsymbol{\theta} = (X^TX)^{-1}X^Ty$ but describes a regression problem, not classification.

- **Logistic Regression**: Adapts linear regression for classification by passing the output through a sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ to model probabilities $p(y|x)$. Parameters are estimated via Maximum Likelihood Estimation (MLE).

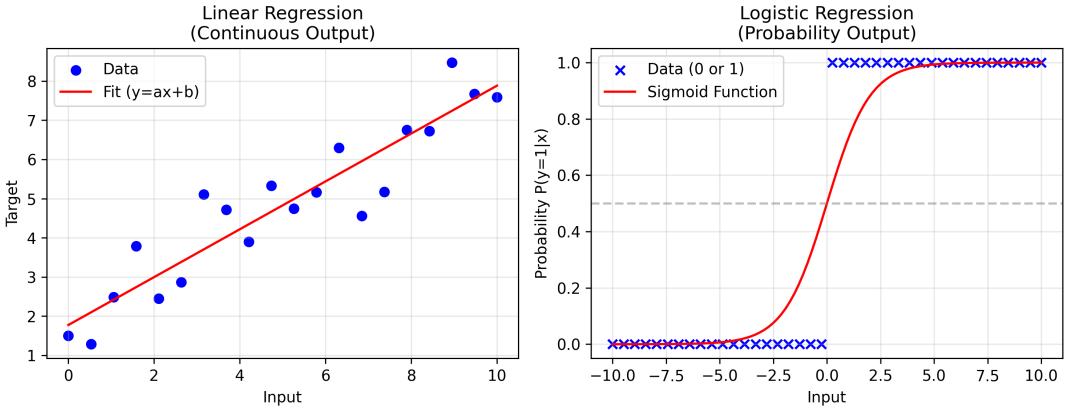


Figure 22: Visual comparison: Linear Regression (left) fits a line to minimize error, while Logistic Regression (right) fits a sigmoid to model class probabilities.

20.2 Generative vs. Discriminative Models

- **Generative Models** (e.g., Naive Bayes, FDA) attempt to model the joint probability $P(X, Y)$ or the class-conditional density $P(X|Y)$ and the prior $P(Y)$. They make assumptions about how the data is generated (e.g., Gaussian distribution). They are often robust to missing data but may perform worse if modeling assumptions are incorrect.

- **Discriminative Models** (e.g., Logistic Regression, SVMs) directly model the posterior $P(Y|X)$ or find a decision boundary that separates classes. They make fewer assumptions about the data distribution and often yield better performance on large datasets.

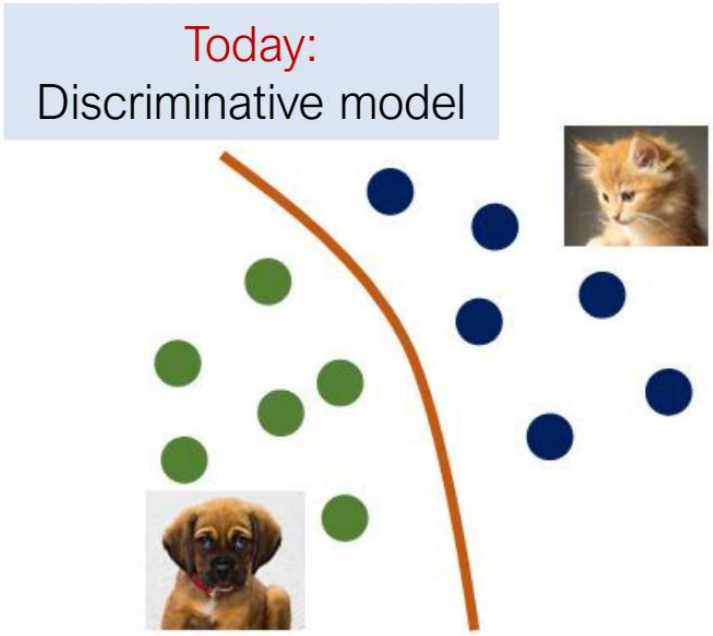


Figure 23: Discriminative models focus on the boundary (left), while Generative models focus on the data distribution (right).

21 Linear SVM: Hard-Margin

21.1 The Linearly Separable Case

Consider a binary classification problem where the training data $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$ is **linearly separable**. Here, $\mathbf{x}^{(i)} \in \mathbb{R}^p$ and $y^{(i)} \in \{-1, +1\}$.

A linear classifier defines a generic hyperplane $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$. The classification rule is:

$$y = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$$

This means we assign class +1 if $\mathbf{w}^T\mathbf{x} + b > 0$ and class -1 otherwise.

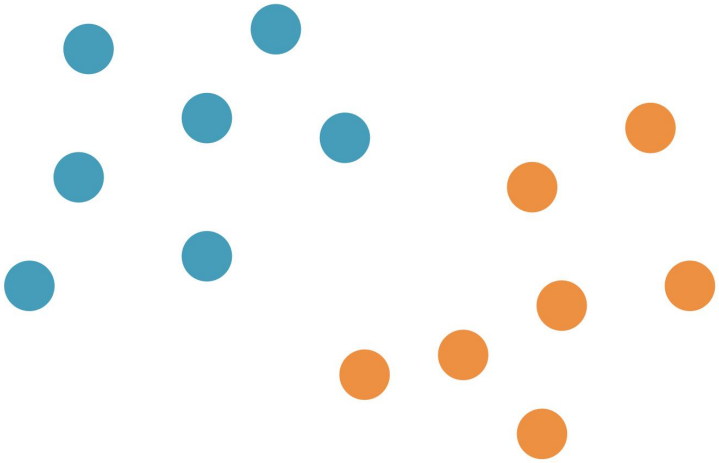


Figure 24: A linearly separable dataset can be separated by a straight line (or hyperplane).

There are infinitely many such hyperplanes that can separate the data. Which one is the best?

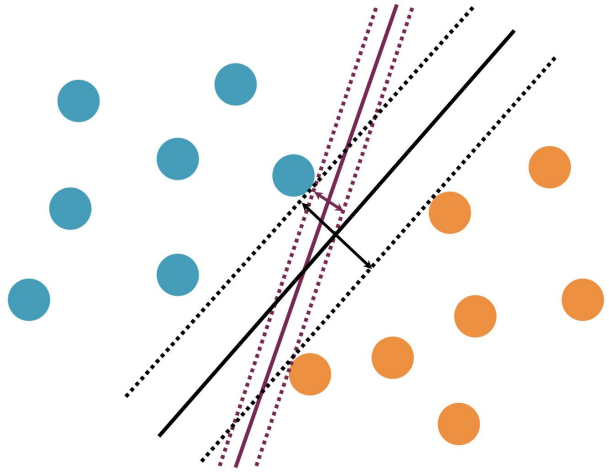


Figure 25: Among multiple separating lines, which one generalizes better?

The **Support Vector Machine (SVM)** essentially chooses the hyperplane that maximizes the **margin**. The margin is defined as the distance between the decision boundary and the nearest training data points (support vectors). A larger margin implies better generalization capability.

21.2 Geometry of the Margin

We define the margin γ as the width of the "street" separating the two classes. We want to maximize this width. Let the decision boundary be defined by $\mathbf{w}^T\mathbf{x} + b = 0$. We can rescale $\mathbf{w}$ and b such that the closest points to the boundary satisfy $|\mathbf{w}^T\mathbf{x} + b| = 1$. These closest points lie on the canoncial hyperplanes:

$$H_1 : \quad \mathbf{w}^T\mathbf{x} + b = +1 \quad \text{(closest positive points)}$$
$$H_2 : \quad \mathbf{w}^T\mathbf{x} + b = -1 \quad \text{(closest negative points)}$$

The margin γ is the perpendicular distance between these two hyperplanes. Let $\mathbf{x}^+$ be a point on H_1 and $\mathbf{x}^-$ be a point on H_2. The normal vector to the hyperplanes is $\frac{\mathbf{w}}{\|\mathbf{w}\|}$. The width is the projection of the vector $(\mathbf{x}^+ - \mathbf{x}^-)$ onto the normal direction:

$$\text{width} = (\mathbf{x}^+ - \mathbf{x}^-) \cdot \frac{\mathbf{w}}{\|\mathbf{w}\|}$$

Using the hyperplane equations:

$$\mathbf{w}^T\mathbf{x}^+ = 1 - b$$
$$\mathbf{w}^T\mathbf{x}^- = -1 - b$$

Substituting these into the width equation:

$$\begin{aligned}\text{width} &= \frac{\mathbf{w}^T\mathbf{x}^+ - \mathbf{w}^T\mathbf{x}^-}{\|\mathbf{w}\|} \\ &= \frac{(1-b) - (-1-b)}{\|\mathbf{w}\|} \\ &= \frac{2}{\|\mathbf{w}\|}\end{aligned}$$

Thus, the margin is $\gamma = \frac{2}{\|\mathbf{w}\|}$.

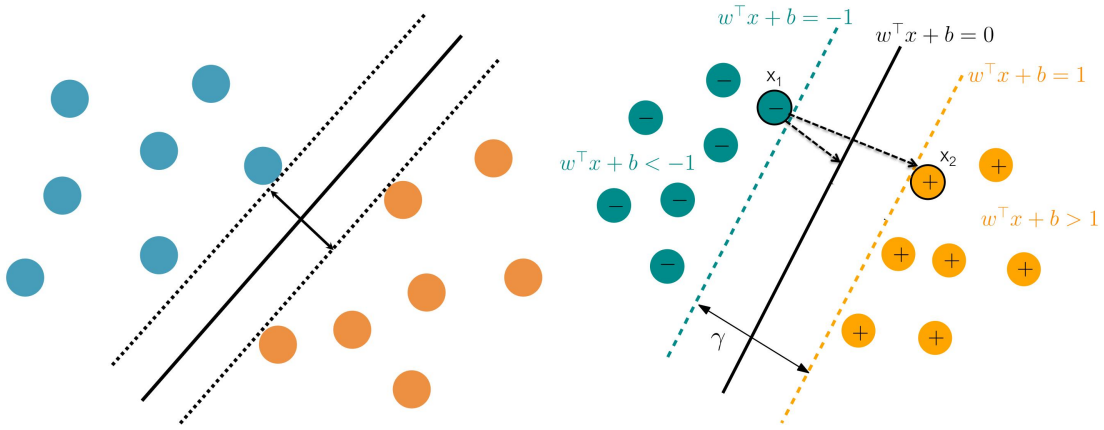


Figure 26: Left: Definition of the margin. Right: To maximize margin $\gamma = 2/\|w\|$, we must minimize $\|w\|$.

21.3 Problem Formulation

To maximize the margin $\frac{2}{\|\mathbf{w}\|}$, we need to minimize $\|\mathbf{w}\|$, or equivalently minimize $\frac{1}{2}\|\mathbf{w}\|^2$ (for mathematical convenience). The optimization must is constrained such that all data points are correctly classified outside the margin.

Primal Optimization Problem:

$$\begin{aligned}\text{minimize} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{subject to} \quad & y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1, \quad i = 1, \ldots, n\end{aligned}$$

The constraint $y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1$ enforces that all positive examples ($y = +1$) have a score ≥ 1 and all negative examples ($y = -1$) have a score ≤ -1.

This is a convex quadratic programming (QP) problem with linear inequality constraints. It guarantees a unique global minimum.

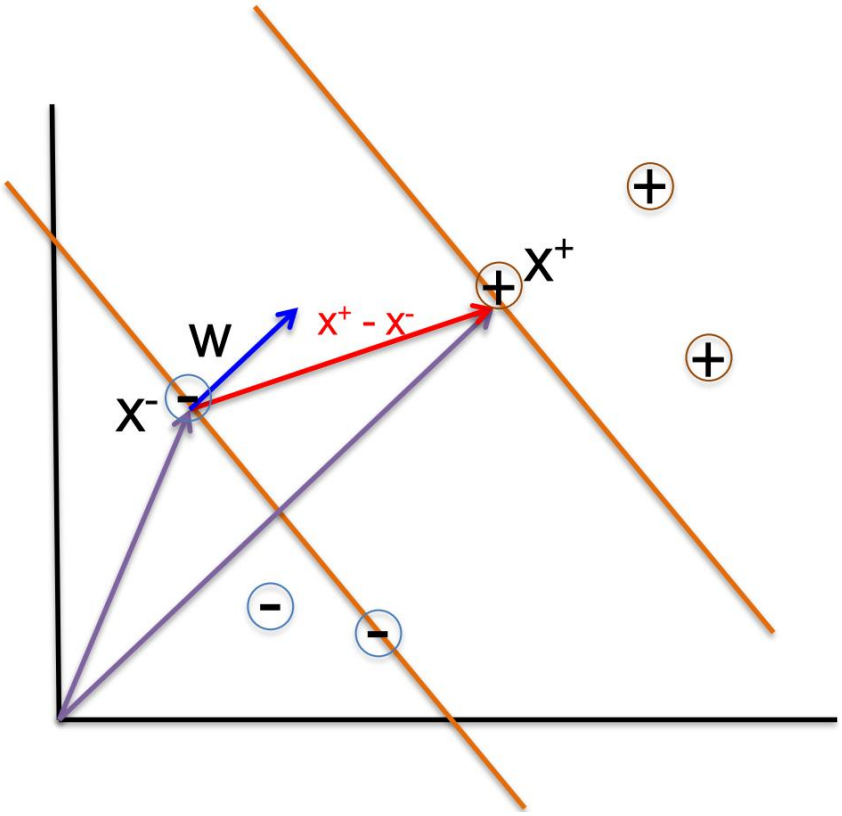


Figure 27: Visualizing the constraints: all points must lie in the "safe" regions (blue and orange), not in the margin.

22 Optimization Framework

22.1 Lagrange Multipliers and the Dual Problem

To solve the constrained optimization problem, we use the method of **Lagrange Multipliers**. This allows us to convert the constrained primal problem into an unconstrained dual problem, which often yields easier solutions and deeper insights (e.g., the importance of support vectors).

The primal problem is:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{subject to} \quad g_i(\mathbf{w},b) = y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) - 1 \geq 0$$

We introduce a Lagrange multiplier $\alpha_i \geq 0$ for each constraint ($i = 1 \ldots n$). The **Lagrangian** function is defined as:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \underbrace{\frac{1}{2}\|\mathbf{w}\|^2}_{\text{Objective}} - \underbrace{\sum_{i=1}^{n} \alpha_i \left(y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) - 1\right)}_{\text{Sum of constraints}}$$

We want to minimize $\mathcal{L}$ with respect to the primal variables $\mathbf{w}, b$ and maximize it with respect to the dual variables $\boldsymbol{\alpha}$.

$$\max_{\boldsymbol{\alpha} \geq 0} \min_{\mathbf{w}, b} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha})$$

22.1.1 Intuition: Forces and Prices

To understand the role of $\boldsymbol{\alpha}$, we can view the optimization physically or economically:

- **Forces**: Imagine the objective function (minimizing $\|\mathbf{w}\|$) as a spring trying to pull the weight vector to zero. The constraints are like walls preventing the points from crossing the margin. α_i represents the "force" that the i-th data point exerts against the wall.
- **Prices**: Alternatively, α_i can be seen as the "price" or "penalty" for violating the i-th constraint. If a constraint is easily satisfied (slack), the price is zero ($\alpha_i = 0$). If a constraint is active (tight), there is a positive cost associated with holding it ($\alpha_i > 0$).

22.2 Deriving the Dual

To find the minimum w.r.t $\mathbf{w}$ and b, we set the gradients to zero:

1. **Derivative w.r.t $\mathbf{w}$**:

$$\nabla_{\mathbf{w}} \mathcal{L} = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y^{(i)} \mathbf{x}^{(i)} = 0 \implies \mathbf{w} = \sum_{i=1}^{n} \alpha_i y^{(i)} \mathbf{x}^{(i)}$$

This tells us that the optimal weight vector $\mathbf{w}$ is a linear combination of the training support vectors.

2. **Derivative w.r.t b**:

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \implies \sum_{i=1}^{n} \alpha_i y^{(i)} = 0$$

Now, we substitute these results back into the Lagrangian to obtain the **Dual Optimization Problem**:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\alpha}) &= \frac{1}{2} \left\| \sum_{i=1}^{n} \alpha_i y^{(i)} \mathbf{x}^{(i)} \right\|^2 - \sum_{i=1}^{n} \alpha_i y^{(i)} \left(\left(\sum_{j=1}^{n} \alpha_j y^{(j)} \mathbf{x}^{(j)} \right)^T \mathbf{x}^{(i)} + b \right) + \sum_{i=1}^{n} \alpha_i \\
&= \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y^{(i)} y^{(j)} (\mathbf{x}^{(i)})^T \mathbf{x}^{(j)} - \sum_{i,j} \alpha_i \alpha_j y^{(i)} y^{(j)} (\mathbf{x}^{(i)})^T \mathbf{x}^{(j)} - b \underbrace{\sum_{i} \alpha_i y^{(i)}}_{0} + \sum_{i} \alpha_i \\
&= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle
\end{aligned}$$

Final Dual Problem:

$$\begin{aligned} \text{maximize} \quad & W(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle \\ \text{subject to} \quad & \alpha_i \geq 0, \quad \forall i \\ & \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \end{aligned}$$

22.3 Support Vectors and Complementary Slackness

The Karush-Kuhn-Tucker (KKT) conditions imply **complementary slackness**:

$$\alpha_i [y^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1] = 0$$

This means that for every data point i:

- Either $\alpha_i = 0$ (the point is correctly classified and outside the margin, irrelevant for the boundary).
- Or $y^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1 = 0 \implies \alpha_i > 0$. These points lie **exactly on the margin** and are called **Support Vectors**.

This sparsity leads to the efficiency of SVMs: the decision boundary is determined *only* by the support vectors.

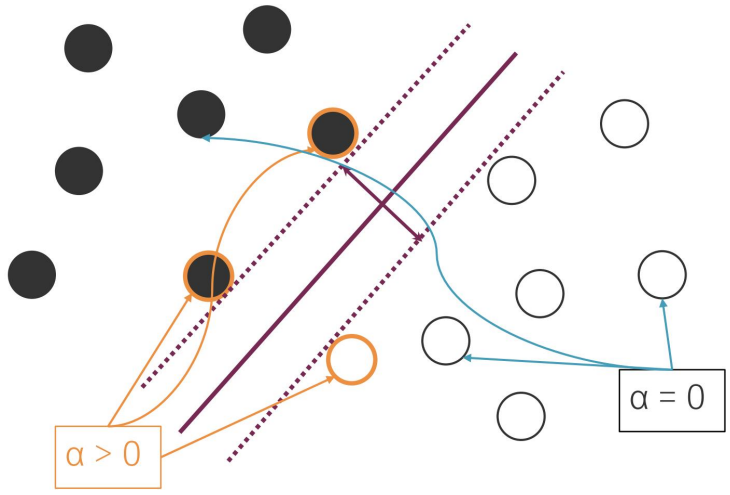


Figure 28: Support vectors (circled) are the only points with $\alpha_i > 0$. They support the margin constraints.

23 Soft-Margin SVM: The Non-Separable Case

23.1 Motivation

In real-world data, classes are rarely perfectly linearly separable due to noise or outliers. Solving the hard-margin problem would result in no solution. We relax the constraints to allow some misclassification while still penalizing it.

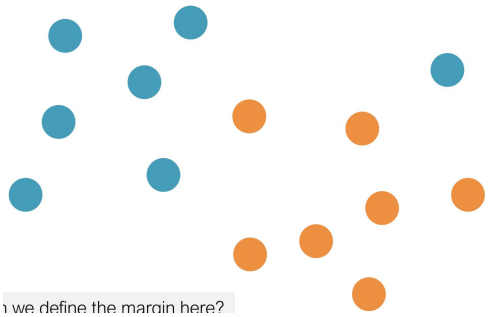


Figure 29: Overlapping classes cannot be separated by a hard margin.

23.2 Slack Variables

We introduce **slack variables** $\xi_i \geq 0$ for each training example. The constraints become:

$$y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi_i$$

- If $\xi_i = 0$, the point is correctly classified and outside the margin (or on it).
- If $0 < \xi_i < 1$, the point is correctly classified but inside the margin (margin violation).
- If $\xi_i > 1$, the point is misclassified.

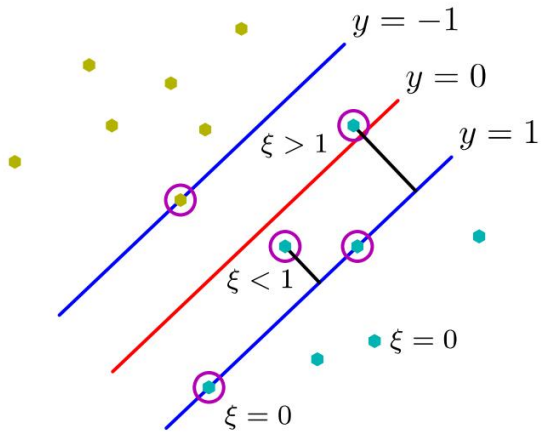


Figure 30: Slack variables ξ_i measure the degree of violation of the margin constraint.

23.3 Soft-Margin Optimization (Primal)

We modify the objective function to minimize both the margin term and the total slack (error):

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$

Subject to:

$$y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

Role of C: C is a regularization hyperparameter balancing margin width vs. classification error.

- **Large C**: Penalizes errors heavily. Harder margin, risk of overfitting.
- **Small C**: Allows more errors. Wider margin, smoother boundary (underfitting risk).

23.4 Hinge Loss Interpretation

The slack variable can be rewritten as $\xi_i = \max(0, 1 - y^{(i)} f(\mathbf{x}^{(i)}))$. This is known as the **Hinge Loss**. The optimization problem is equivalent to minimizing:

$$J(\mathbf{w}, b) = \underbrace{\frac{1}{2}\|\mathbf{w}\|^2}_{\text{Regularization}} + C \sum_{i=1}^{n} \underbrace{\max(0, 1 - y^{(i)} f(\mathbf{x}^{(i)}))}_{\text{Hinge Loss}}$$

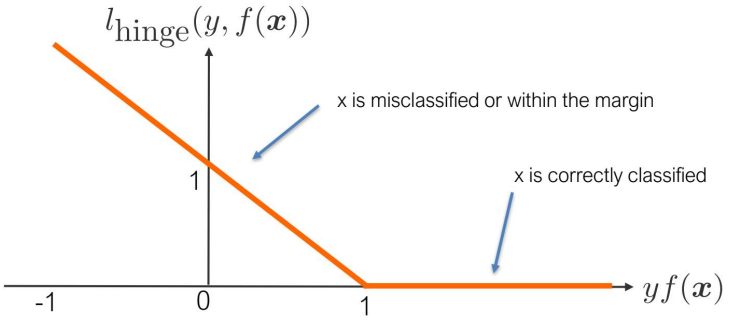


Figure 31: The Hinge Loss function is 0 if the point is correctly classified with margin ≥ 1, and increases linearly otherwise.

23.5 Soft-Margin Dual Problem

The derivation of the dual is almost identical to the hard-margin case, with one key difference in results (due to the constraints $\xi_i \geq 0$ and the upper bound on α from the Lagrangian). The Dual problem becomes:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \quad \forall i \\ & \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \end{aligned}$$

The condition $0 \leq \alpha_i \leq C$ is the only change! This is colloquially known as the "box constraint". Support vectors now include points on the margin ($0 < \alpha_i < C$) and violators ($\alpha_i = C$).

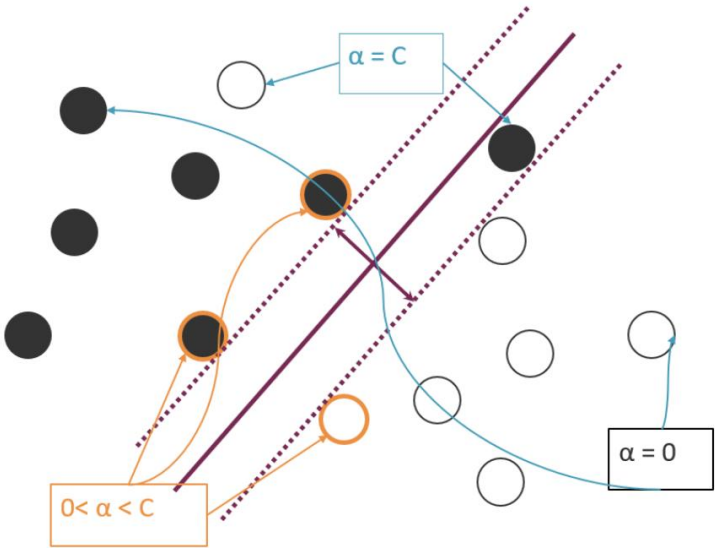


Figure 32: Summary of the Dual Optimization Problem.

24 Kernel Methods: The Non-Linear Case

24.1 The Need for Non-Linearity

Linear SVMs are powerful, but many real-world datasets are not linearly separable, even with soft margins. The decision boundary might be a curve or a complex shape.

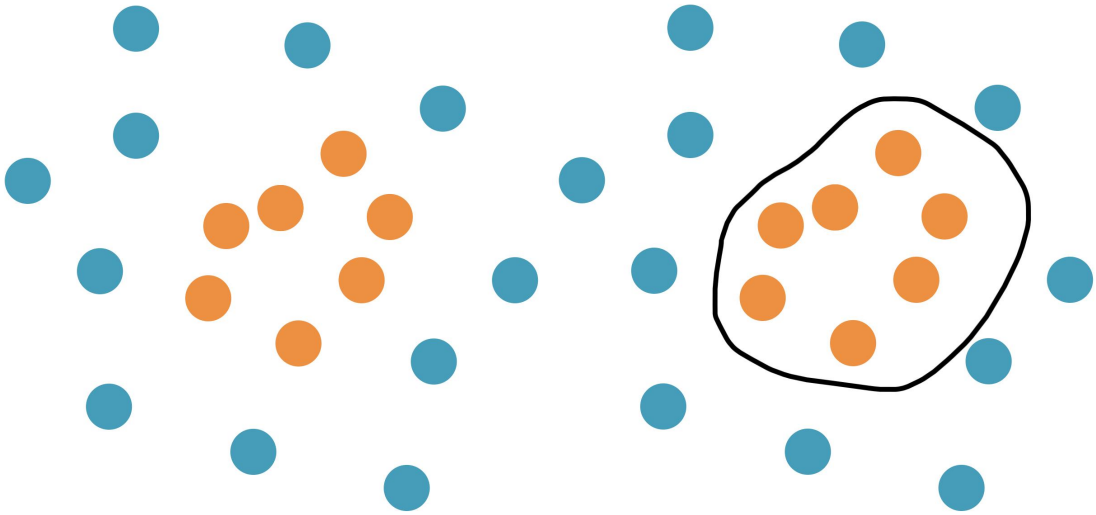


Figure 33: Datasets requiring non-linear decision boundaries.

24.2 Feature Mapping

The core idea is to map the input data $\mathbf{x}$ from the original input space (where it is not linearly separable) to a higher-dimensional **feature space** where it becomes linearly separable. Let $\Phi : \mathbb{R}^p \to \mathcal{H}$ be a non-linear mapping. We replace every occurrence of $\mathbf{x}$ with $\Phi(\mathbf{x})$. The linear decision boundary in $\mathcal{H}$, $\mathbf{w}^T\Phi(\mathbf{x}) + b = 0$, corresponds to a non-linear boundary in the original space.

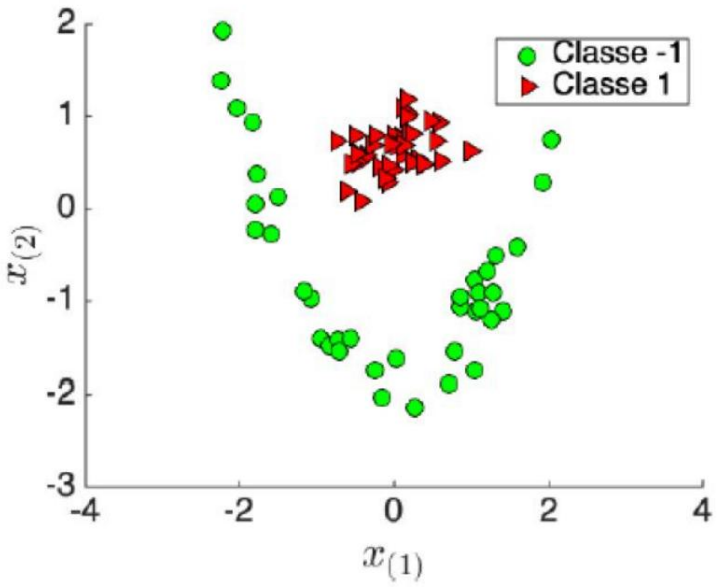


Figure 34: Mapping data to a higher dimension can make it linearly separable.

24.3 The Kernel Trick

Explicitly computing the mapping $\Phi(\mathbf{x})$ can be computationally expensive or even impossible (if $\mathcal{H}$ is infinite-dimensional). However, notice that in the Dual SVM formulation, the data points only appear in inner products: $\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$. In the transformed space, we need $\langle \Phi(\mathbf{x}^{(i)}), \Phi(\mathbf{x}^{(j)}) \rangle$.

The Kernel Trick: We define a **Kernel Function** $K(\mathbf{x}, \mathbf{y})$ such that:

$$K(\mathbf{x}, \mathbf{y}) = \langle \Phi(\mathbf{x}), \Phi(\mathbf{y}) \rangle$$

This allows us to compute the inner product in the high-dimensional space *without ever computing the coordinates* $\Phi(\mathbf{x})$.

The Dual Optimization problem using Kernels becomes:

$$\text{maximize} \quad W(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$$

Derivation of the Decision Function: The linear decision boundary in the feature space is $f(\mathbf{x}) = \mathbf{w}^T \Phi(\mathbf{x}) + b$. From the dual formulation, we know that the optimal weight vector is a linear combination of the support vectors in the feature space:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y^{(i)} \Phi(\mathbf{x}^{(i)})$$

Substituting this back into the decision function:

$$f(\mathbf{x}) = \left(\sum_{i=1}^{n} \alpha_i y^{(i)} \Phi(\mathbf{x}^{(i)}) \right)^T \Phi(\mathbf{x}) + b = \sum_{i=1}^{n} \alpha_i y^{(i)} \langle \Phi(\mathbf{x}^{(i)}), \Phi(\mathbf{x}) \rangle + b$$

By applying the kernel definition $K(\mathbf{x}^{(i)}, \mathbf{x}) = \langle \Phi(\mathbf{x}^{(i)}), \Phi(\mathbf{x}) \rangle$, we obtain the final formula.

Predictions are made using:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i y^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}) + b$$

The final classification is simply the sign of $f(x)$:

$$\hat{y} = \text{sign}(f(\mathbf{x})) = \begin{cases} +1 & \text{if } f(\mathbf{x}) > 0 \\ -1 & \text{if } f(\mathbf{x}) < 0 \end{cases}$$

24.4 Common Kernels

24.4.1 Polynomial Kernel

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y} + c)^d$$

where d is the degree of the polynomial.

24.4.2 Radial Basis Function (RBF) / Gaussian Kernel

The most popular kernel. It corresponds to an infinite-dimensional feature space.

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2}\right) = \exp(-\gamma \|\mathbf{x} - \mathbf{y}\|^2)$$

The parameter σ (or γ) controls the width of the Gaussian.

- **Small σ (Large γ)**: The kernel is very peaked. Each point has local influence. High risk of overfitting.
- **Large σ (Small γ)**: The kernel is smooth. Points influence distant neighbors. Risk of underfitting (approaches linear).

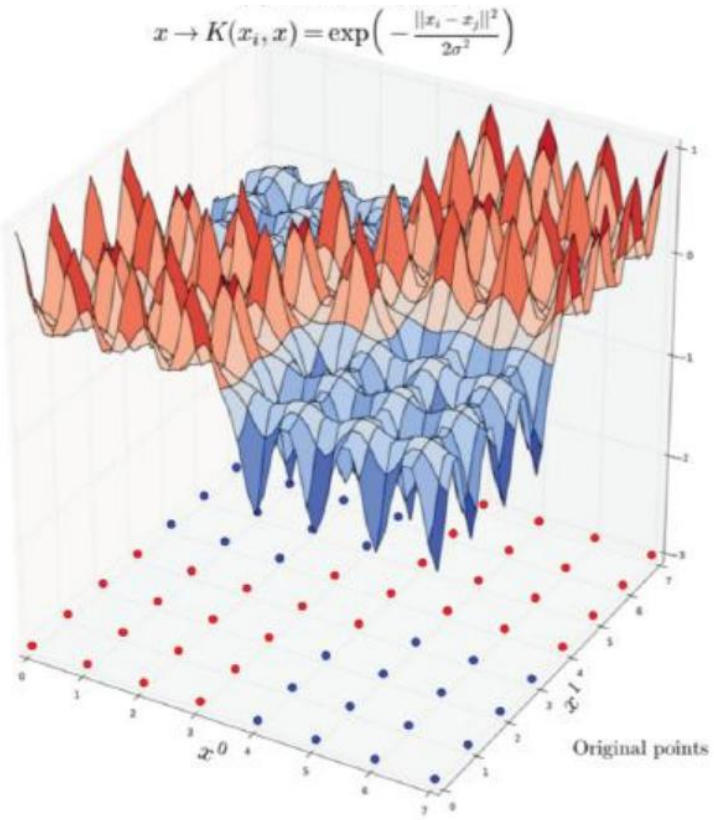


Figure 35: Effect of RBF Kernel bandwith on the decision boundary.

25 Lab Work Summary

In the accompanying lab, we applied SVMs to both synthetic and real-world datasets to observe the effects of hyperparameters and kernels.

25.1 SVM with Linear Kernel

We implemented a linear kernel $K(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{y}$ to classify a linearly separable dataset containing an outlier.

- **Effect of C**: We observed that a high value ($C = 100$) forced the model to correctly classify the outlier, resulting in a narrow margin and potential overfitting. A lower value ($C = 1$) allowed the outlier to be misclassified (treated as noise), leading to a wider margin and a more natural decision boundary.

25.2 SVM with Gaussian Kernel

We explored non-linear classification using the Gaussian (RBF) kernel:

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2}\right)$$

We analyzed the interplay between C and σ. A small σ creates complex, island-like boundaries (overfitting), while a large σ smooths the boundary.

25.3 Text Categorization (Sentiment Analysis)

We applied SVMs to a text mining problem: identifying positive vs. negative movie reviews. **Pipeline**:

1. **Preprocessing**: Conversion to lowercase, removal of punctuation and stop words, and stemming (reducing words to their roots).
2. **Feature Extraction**: Using the **TF-IDF** (Term Frequency-Inverse Document Frequency) method to represent documents as vectors, weighing words by their identifying power.
3. **Classification**: Training a `LinearSVC` on the TF-IDF matrix.
4. **Evaluation**: Performance was measured using Precision, Recall, and F1-score.

26 Summary

- **Hard-Margin SVM**: Maximizes margin $2/\|\mathbf{w}\|$ for linearly separable data.
- **Soft-Margin SVM**: Introduces slack variables ξ_i and parameter C to handle non-separable data and outliers.
- **Kernel SVM**: Uses the kernel trick to implicitly map data to high-dimensional spaces, enabling non-linear classification efficiently.

Lecture 5: Neural Networks

Contents

27 Introduction

This document serves as a comprehensive study resource for Lecture 5, focusing on **Dimensionality Reduction**. Before delving into the core topic of the day, we briefly review the concepts from the previous lecture concerning Support Vector Machines (SVMs) and Kernel Methods.

The lecture is structured as follows:

1. **Recap**: Discriminative vs. Generative models and Support Vector Machines (SVMs).
2. **The Kernel Trick**: Extending linear models to non-linear problems.
3. **Dimensionality Reduction**:
    - Singular Value Decomposition (SVD).
    - Principal Component Analysis (PCA).
4. **Non-Linear Embeddings**: Multidimensional Scaling (MDS), Isomap, and Locally Linear Embedding (LLE).
5. **Feature Selection**: Methods for selecting relevant subsets of features.

28 Review: Support Vector Machines (SVMs)

28.1 Discriminative vs. Generative Models

In supervised learning, we distinguish between two primary approaches for modeling the relationship between input data $\mathbf{x}$ and labels y:

- **Discriminative Models** learn the decision boundary directly by modeling the conditional probability $P(y|\mathbf{x})$. They do not model how the data is generated but focus solely on separating the classes.

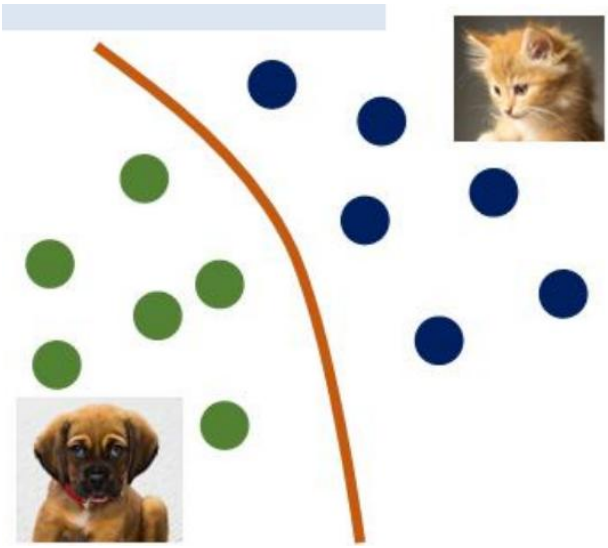


Figure 36: Discriminative Model: Focuses on finding the boundary that separates the classes.

- **Generative Models** learn the joint probability distribution $P(\mathbf{x}, y)$ (often by modeling $P(\mathbf{x}|y)$ and $P(y)$). They describe how the data is generated and can produce new data points.

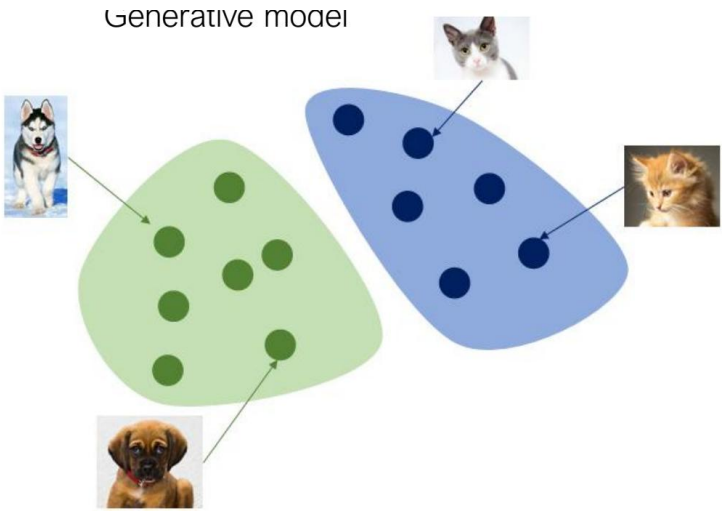


Figure 37: Generative Model: Focuses on describing the distribution of each class.

SVMs fall into the **discriminative** category. They are robust to model mis-specification and typically perform well on large datasets where defining a precise generative process is difficult.

28.2 Hard Magnitude and Soft Margin SVMs

The fundamental idea of an SVM is to find the hyperplane that separates two classes with the *maximum margin*. The margin is defined as the distance between the decision boundary and the nearest data points (support vectors).

The optimization problem for a Hard Margin SVM (linearly separable data) is:

$$\min_{\mathbf{w},b} \quad \frac{1}{2}\|\mathbf{w}\|^2 \tag{1}$$

subject to:

$$y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1, \quad \forall i = 1, \ldots, n \tag{2}$$

By introducing Lagrange multipliers $\alpha_i \geq 0$, we obtain the **Dual Formulation**:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle \tag{3}$$

subject to $\sum_{i=1}^{n} \alpha_i y^{(i)} = 0$ and $\alpha_i \geq 0$.

Key Observation: The dual problem depends on the data samples $\mathbf{x}$ *only* through their inner products $\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$. This property is the foundation of the Kernel Trick.

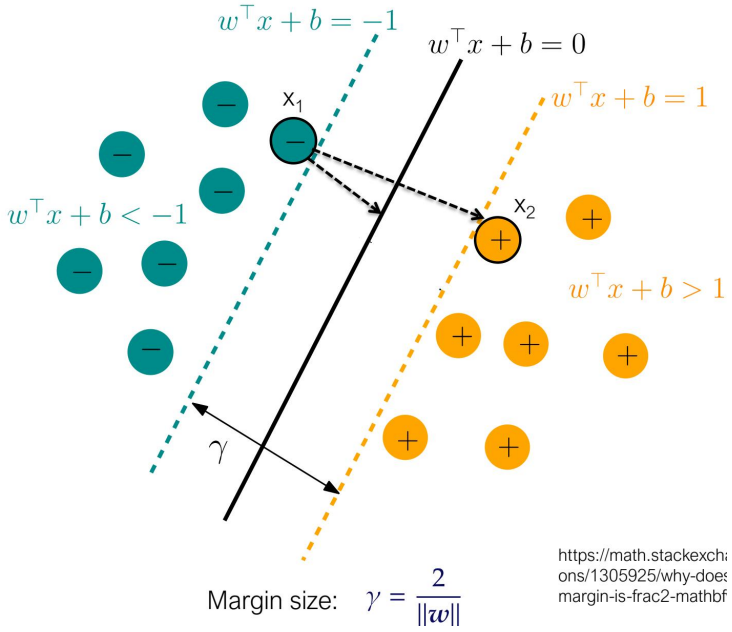
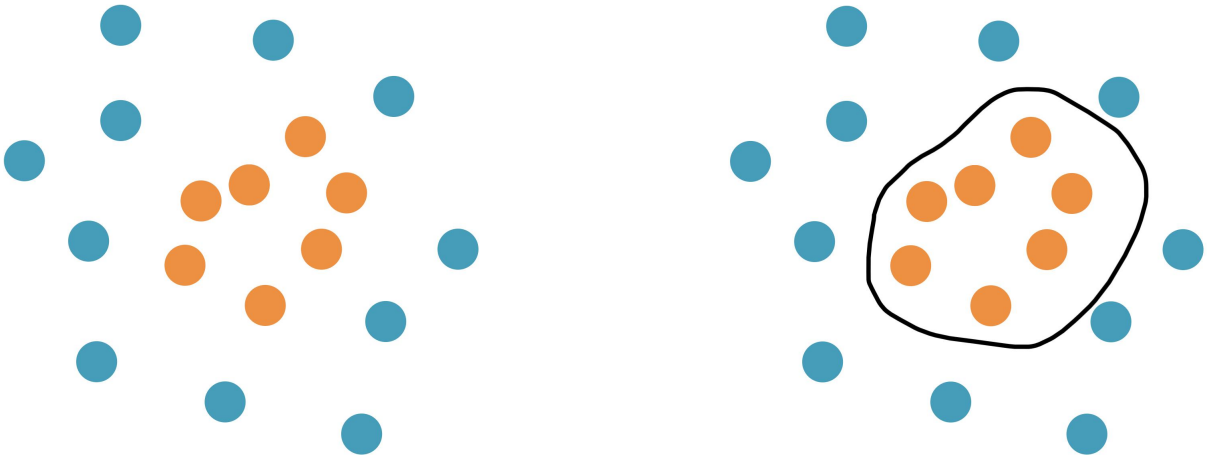


Figure 38: SVM Decision Boundary and Margin.

28.3 The Kernel Trick

What if the data is not linearly separable?



(a) Non-linear boundary needed.

(b) Another complex distribution.

Figure 39: Examples of non-linearly separable data.

The **Kernel Trick** allows us to apply linear methods (like SVMs) to non-linear problems by mapping the input data into a higher-dimensional feature space $\mathcal{H}$ where specific patterns become linearly separable.

Let $\Phi : \mathcal{X} \to \mathcal{H}$ be a mapping function. We replace every occurrence of the inner product $\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$ in the dual formulation with the inner product in the feature space:

$$\langle \Phi(\mathbf{x}^{(i)}), \Phi(\mathbf{x}^{(j)}) \rangle \tag{4}$$

Computing $\Phi(\mathbf{x})$ explicitly can be computationally expensive or infinite-dimensional. However, a **Kernel Function** $K(\mathbf{x}, \mathbf{y})$ allows us to compute this inner product directly

in the original space:

$$K(\mathbf{x}, \mathbf{y}) = \langle \Phi(\mathbf{x}), \Phi(\mathbf{y}) \rangle \tag{5}$$

28.3.1 Common Kernels

1. **Polynomial Kernel**:

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y} + c)^d \tag{6}$$

   Maps data to a space of polynomial combinations of features up to degree d.

2. **Gaussian (RBF) Kernel**:

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2}\right) \tag{7}$$

   This maps data into an infinite-dimensional space. The parameter σ (bandwidth) controls the "reach" of a single training example.

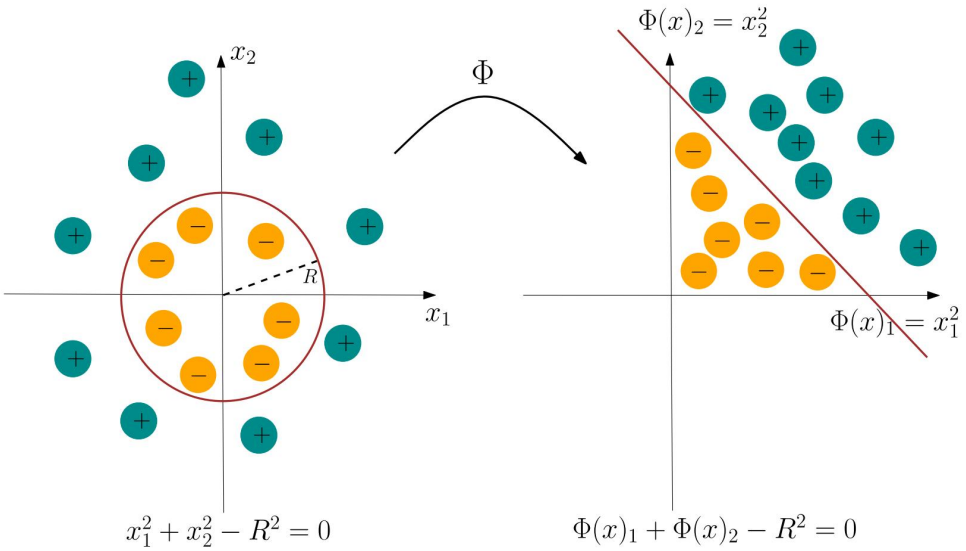
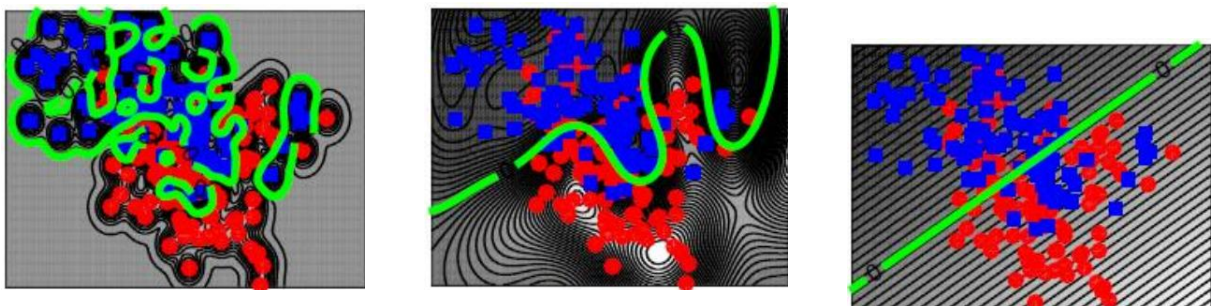


Figure 40: Visualizing the transformation: A circle in 2D becomes a plane cut in 3D via $\Phi((x_1, x_2)) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$.

28.3.2 Effect of Kernel Parameters

The choice of hyperparameters (like σ for RBF) is critical.



(a) Small σ: Overfitting (islands).

(b) Good σ: Smooth boundary.

(c) Large σ: Underfitting.

Figure 41: Influence of the Gaussian Kernel bandwidth σ.

29 Dimensionality Reduction

29.1 Motivation: The Curse of Dimensionality

In modern machine learning, we often deal with high-dimensional data (e.g., images with megapixels, text with vocabulary size of thousands). High dimensionality introduces several problems, collectively known as the *Curse of Dimensionality*:

1. **Sparsity**: As dimensions increase, data becomes incredibly sparse. To maintain the same data density, the number of samples required grows exponentially with dimension.
2. **Distance Concentration**: In very high dimensions, the distance between the nearest and farthest points tends to converge, making distance-based algorithms (like k-NN or K-Means) ineffective.
3. **Computational Cost**: Storage and processing complexity often scale with d.

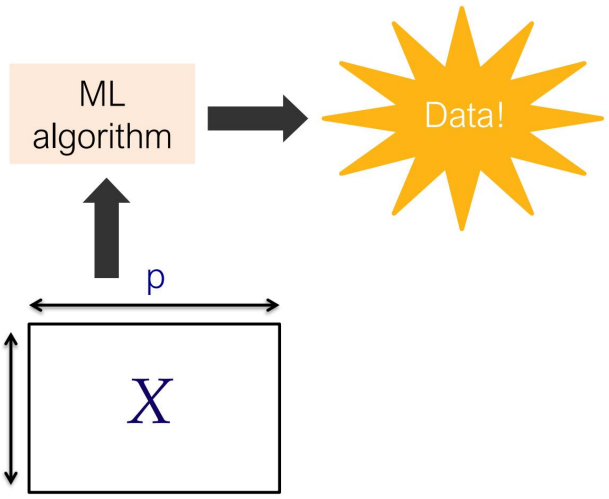


Figure 42: Key issues with high dimensionality: Sparsity and Computational Complexity.

29.2 Intrinsic Dimensionality and Rank

Often, high-dimensional data lies on a lower-dimensional manifold. For example, points on a line in 2D space effectively have 1 dimension, even if represented by (x, y) coordinates.

Rank of a Matrix Consider a data matrix $\mathbf{A}$ where each row is a sample. The **rank** of $\mathbf{A}$ is the number of linearly independent rows (or columns). It represents the true "information content" dimensionality. If $\text{rank}(\mathbf{A}) < \min(m, n)$, the data is redundant and can be compressed without error.

29.3 Signal vs. Noise

Real world data often follows:

$$\text{Data} = \text{Signal} + \text{Noise} \tag{8}$$

Dimensionality reduction aims to approximate the useful signal in a lower-dimensional space, effectively filtering out the noise which often manifests in the "extra" dimensions.

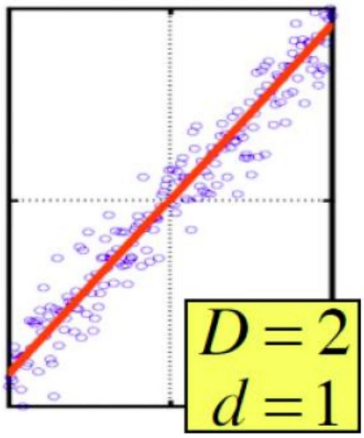


Figure 43: Signal vs Noise: The data points (blue crosses) deviate slightly from the underlying linear relationship (red line) due to noise. Using 1 dimension (the line) captures the signal.

30 Singular Value Decomposition (SVD)

SVD is a fundamental matrix factorization technique that generalizes eigendecomposition to non-square matrices. It is the algebraic foundation for Principal Component Analysis (PCA).

30.1 Definition

Any real matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ can be factored as:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{9}$$

where:

- $\mathbf{U} \in \mathbb{R}^{m \times m}$ is an orthogonal matrix ($\mathbf{U}^T\mathbf{U} = \mathbf{I}$). Its columns are the **left singular vectors** (eigenvectors of $\mathbf{A}\mathbf{A}^T$).
- $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ is a diagonal matrix with non-negative entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ on the diagonal. These are the **singular values** (square roots of eigenvalues of $\mathbf{A}^T\mathbf{A}$).
- $\mathbf{V} \in \mathbb{R}^{n \times n}$ is an orthogonal matrix. Its columns are the **right singular vectors** (eigenvectors of $\mathbf{A}^T\mathbf{A}$).

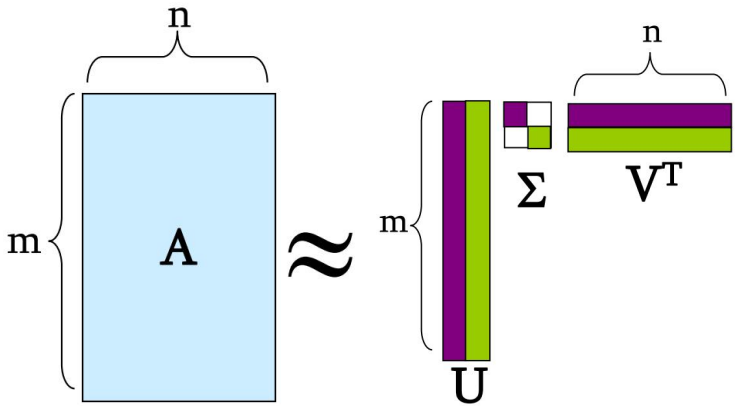


Figure 44: Visual representation of SVD decomposition.

30.2 Low-Rank Approximation

One of the most powerful applications of SVD is constructing a low-rank approximation of a matrix. By keeping only the top k singular values and setting the rest to zero, we obtain the "best" rank-k approximation $\mathbf{A}_k$ in terms of the Frobenius norm (Eckart-Young-Mirsky Theorem).

$$\mathbf{A}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \tag{10}$$

This effectively compresses the matrix by retaining the strongest "concepts" (singular values) and discarding the weaker ones (noise).

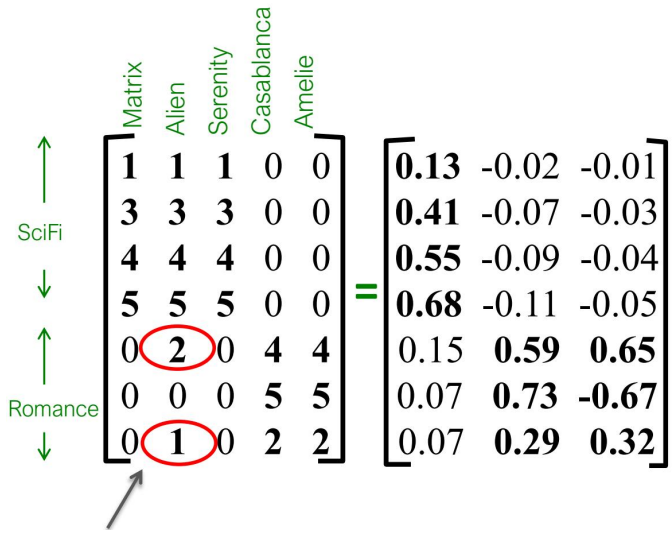


Figure 45: SVD separating Signal (User concepts) from Noise in a Recommender System context.

30.3 Choosing the Rank k

How many dimensions/singular values should we keep? A common heuristic is to preserve a certain percentage (e.g., 90%) of the total variance (energy), defined by the sum of

squared singular values:

$$\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{j=1}^{\min(m,n)} \sigma_j^2} \geq 0.90 \tag{11}$$

31 Principal Component Analysis (PCA)

PCA is arguably the most popular dimensionality reduction technique. It is a direct application of SVD but is often derived from a statistical perspective: finding the directions (principal components) that maximize the variance of the data.

31.1 Standardization

Before applying PCA, it is crucial to standardize the data, as PCA is sensitive to the scale of features. For each feature j:

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j} \tag{12}$$

where μ_j is the mean and σ_j is the standard deviation. This ensures Mean(Data) $= 0$ and Var(Data) $= \mathbf{I}$.

31.2 Variance Maximization: The Derivation

We seek a unit vector $\mathbf{w}$ such that the projection of the data $\mathbf{X}$ onto $\mathbf{w}$ has maximal variance. The projection of a data point $\mathbf{x}$ onto $\mathbf{w}$ is $z = \mathbf{w}^T\mathbf{x}$. The variance of the projected data is:

$$\text{Var}(\mathbf{z}) = \text{Var}(\mathbf{w}^T\mathbf{X}^T) = \mathbf{w}^T\text{Cov}(\mathbf{X})\mathbf{w} \tag{13}$$

Let $\mathbf{S} = \frac{1}{m-1}\mathbf{X}^T\mathbf{X}$ be the sample covariance matrix (assuming centered data). We want to maximize:

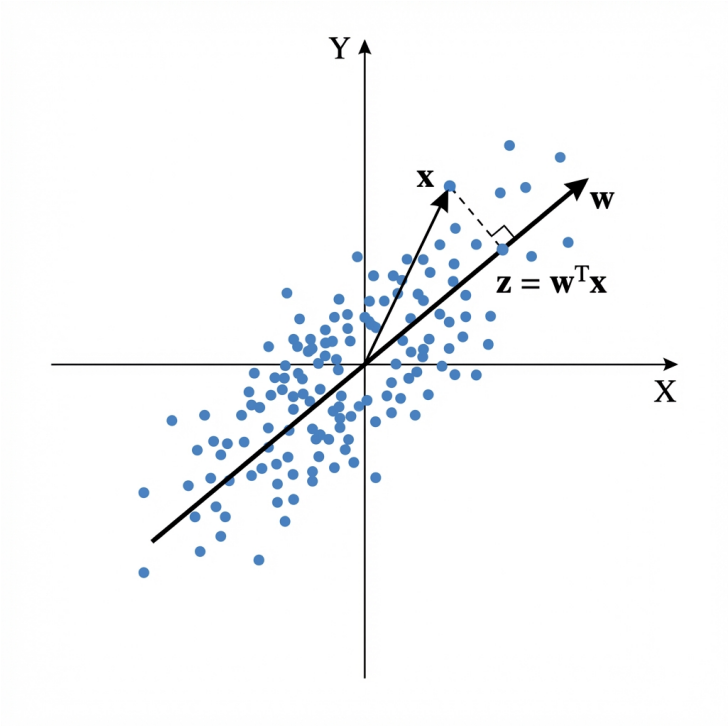


Figure 46: Projection of a data point $\mathbf{x}$ onto the direction $\mathbf{w}$. The goal is to maximize the variance of the projections z.

$$\max_{\mathbf{w}} \mathbf{w}^T \mathbf{S} \mathbf{w} \quad \text{subject to} \quad \|\mathbf{w}\|^2 = 1 \tag{14}$$

To solve this constrained optimization problem, we use the method of Lagrange Multipliers. The Lagrangian is:

$$\mathcal{L}(\mathbf{w}, \lambda) = \mathbf{w}^T \mathbf{S} \mathbf{w} - \lambda(\mathbf{w}^T \mathbf{w} - 1) \tag{15}$$

We take the derivative with respect to $\mathbf{w}$ and set it to zero:

$$\nabla_{\mathbf{w}} \mathcal{L} = 2\mathbf{S}\mathbf{w} - 2\lambda \mathbf{w} = 0 \tag{16}$$

$$\mathbf{S}\mathbf{w} = \lambda \mathbf{w} \tag{17}$$

Conclusion: The direction $\mathbf{w}$ that maximizes variance is an **eigenvector** of the covariance matrix $\mathbf{S}$. The variance itself is $\mathbf{w}^T \mathbf{S} \mathbf{w} = \mathbf{w}^T(\lambda \mathbf{w}) = \lambda \|\mathbf{w}\|^2 = \lambda$. Thus, to maximize variance, we must choose the eigenvector corresponding to the **largest eigenvalue** λ.

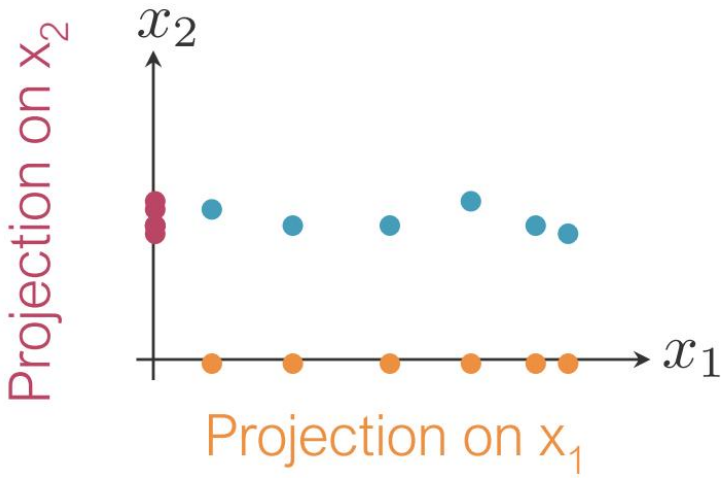


Figure 47: PCA finds the axis of maximal variance.

31.3 The PCA Algorithm

1. Standardize the data $\mathbf{X}$.
2. Compute the covariance matrix $\mathbf{\Sigma} \approx \frac{1}{m-1}\mathbf{X}^T\mathbf{X}$.
3. Compute eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_n$ and eigenvalues $\lambda_1, \ldots, \lambda_n$ of $\mathbf{\Sigma}$.
4. Sort eigenvalues in descending order $\lambda_1 \geq \lambda_2 \ldots$.
5. Select the top k eigenvectors $\mathbf{U}_k = [\mathbf{u}_1, \ldots, \mathbf{u}_k]$.
6. Project the data: $\mathbf{X}_{new} = \mathbf{X}\mathbf{U}_k$.

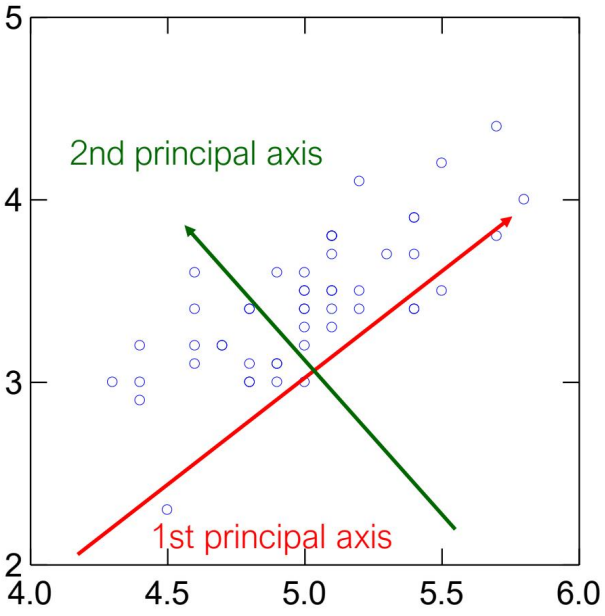


Figure 48: First and second principal components.

31.4 Choosing Dimensions

Similar to SVD, we inspect the cumulative explained variance.

$$\text{Ratio} = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{j=1}^{n} \lambda_j} \tag{18}$$

We typically retain enough components to explain 85-95% of the variance.

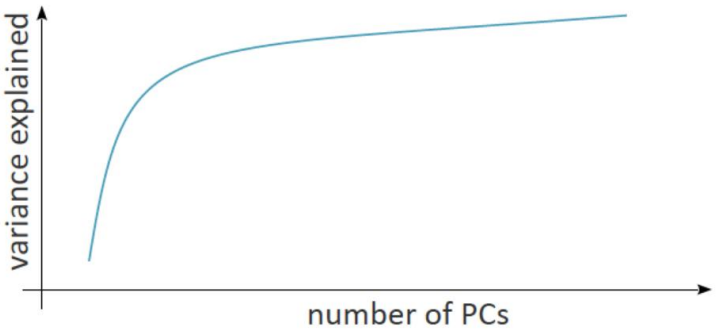


Figure 49: Explained variance ratio as a function of the number of components.

32 Non-Linear Dimensionality Reduction

Linear methods like PCA and SVD fail to capture the structure of data lying on non-linear manifolds (e.g., a swiss roll). Non-linear techniques address this by preserving different properties of the data.

32.1 Multidimensional Scaling (MDS)

MDS aims to map data to a lower-dimensional space while preserving the **pairwise distances** between points. It minimizes the *stress function*:

$$\text{Stress} = \sqrt{\frac{\sum_{i,j}(d_{ij} - \delta_{ij})^2}{\sum_{i,j} \delta_{ij}^2}} \tag{19}$$

where δ_{ij} are the distances in the original space and d_{ij} are distances in the embedded space.

The resulting **embedded space** is a lower-dimensional coordinate system where the Euclidean average distances between points approximate closer as possible to the original dissimilarities. In this new space, the specific coordinates might not have an inherent physical meaning (unlike the original features), but the relative positions preserve the structural relationships of the dataset. For instance, if MDS is applied to a matrix of flight distances between cities, the resulting 2D plot effectively reconstructs the map of the cities' locations.

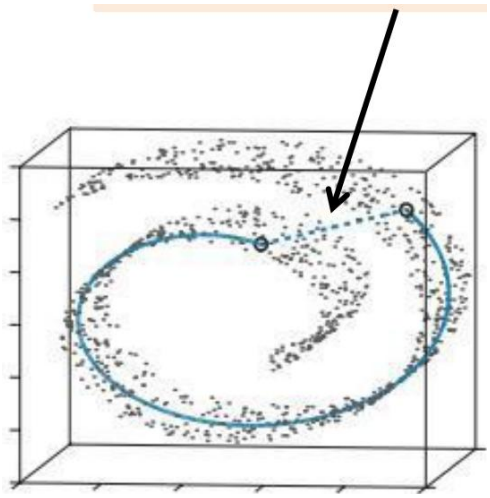
32.2 Isomap

Isomap extends MDS by using **Geodesic distances** instead of Euclidean distances.

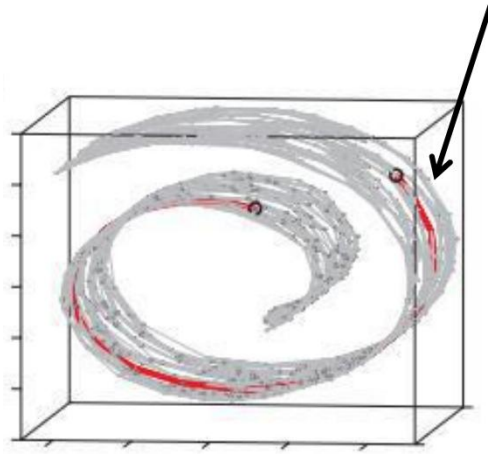
1. Construct a neighborhood graph (e.g., connects each point to its k nearest neighbors).

2. Compute shortest paths between all pairs of nodes (e.g., Dijkstra's algorithm). This approximates the geodesic distance on the manifold.

3. Apply MDS using these geodesic distances.



(a) Euclidean distance (dashed) vs Manifold distance (solid).



(b) Neighborhood graph on Swiss Roll.

Figure 50: Isomap concept: Measuring distance along the manifold.

32.3 Locally Linear Embedding (LLE)

LLE focuses on preserving local neighborhood geometry rather than global distances.

1. Find k-nearest neighbors for every point.

2. Compute weights W_{ij} to reconstruct each point linearly from its neighbors.

3. Find low-dimensional coordinates Y that are best reconstructed by the *same* weights.

$$\min_{Y} \sum_{i} \left\| \mathbf{y}_i - \sum_{j} W_{ij} \mathbf{y}_j \right\|^2 \tag{20}$$

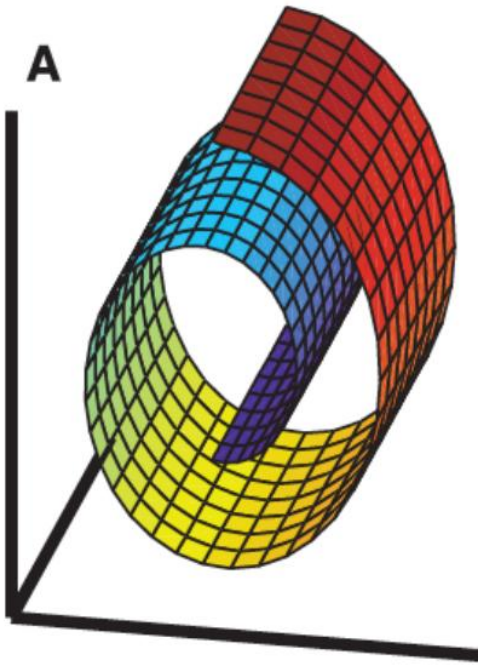


Figure 51: LLE unrolling the Swiss Roll manifold.

33 Feature Selection

Unlike dimensionality reduction which creates *new* features (combinations of old ones), feature selection chooses a subset of the *original* features.

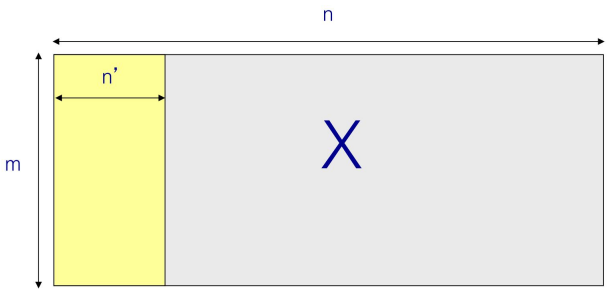


Figure 52: Selecting relevant features.

33.1 Techniques

- **Filter Methods**: Rank features by a statistical score (e.g., Correlation, χ^2 test) independent of the model. Fast but ignores feature interactions.
- **Wrapper Methods**: Search for the best subset by training a model (e.g., Forward Selection, Backward Elimination). Accurate but computationally expensive ($O(n^2)$).
- **Embedded Methods**: Feature selection is part of the training process (e.g., Lasso regularization).

34 Lab Session: Dimensionality Reduction

In this session, we apply the concepts of SVD, PCA, and MDS to practical problems involving image compression, spectral data analysis, and map reconstruction.

34.1 Image Compression using SVD

We analyze how SVD can compress an image by retaining only the most significant singular values.

1. **Method**: An image $\mathbf{X}$ is decomposed as $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. The low-rank approximation $\mathbf{X}_k$ uses only the top k singular values.
2. **Observation**:
   - At low k (e.g., $k = 10$), the image is blurry but recognizable (major structure preserved).
   - As k increases ($k = 50, 100$), details return.
   - The reconstruction error $\|\mathbf{X} - \mathbf{X}_k\|_F$ decreases monotonically with k.

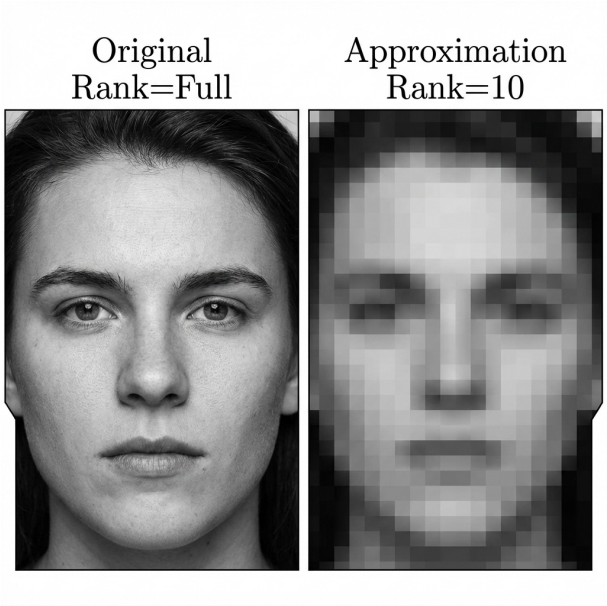


Figure 53: Effect of Rank on Image Reconstruction (SVD).

34.2 PCA on the Wine Dataset

We use PCA to visualize a 13-dimensional dataset (chemical features of wines) in 2D and 3D.

1. **Variance Analysis**: The first few principal components capture the majority of the variance (information).
2. **Projection**: Plotting the data on the first two principal components often reveals distinct clusters corresponding to different wine cultivars, even without using labels during training. This demonstrates PCA's ability to uncover latent structure.

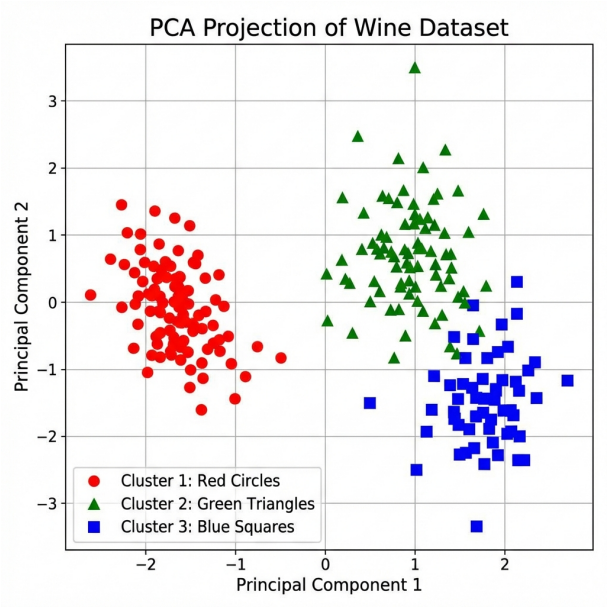


Figure 54: PCA projection of the Wine dataset revealing latent clusters.

34.3 Map Reconstruction with MDS

Given a matrix of pairwise distances between US cities, we use MDS to recover their relative coordinates. The process involves **Classical MDS** (or PCoA):

1. **Distance Matrix**: Let $\mathbf{D}$ be the $m \times m$ matrix of squared pairwise distances $D_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|^2$.

2. **Double Centering**: Compute the **Centering Matrix J**.

$$\mathbf{J} = \mathbf{I} - \frac{1}{m}\mathbf{1}\mathbf{1}^T \tag{21}$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{1}$ is a vector of ones. $\mathbf{J}$ is a projection matrix that subtracts the mean. Apply $\mathbf{J}$ to $\mathbf{D}$ to obtain the Gram matrix $\mathbf{B}$:

$$\mathbf{B} = -\frac{1}{2}\mathbf{J}\mathbf{D}\mathbf{J} \tag{22}$$

Intuition: Multiplying by $\mathbf{J}$ on both sides performs **double centering**: it subtracts the row means and the column means. This centers the data at the origin and converts the squared distances back into inner products ($\mathbf{B} \approx \mathbf{X}\mathbf{X}^T$).

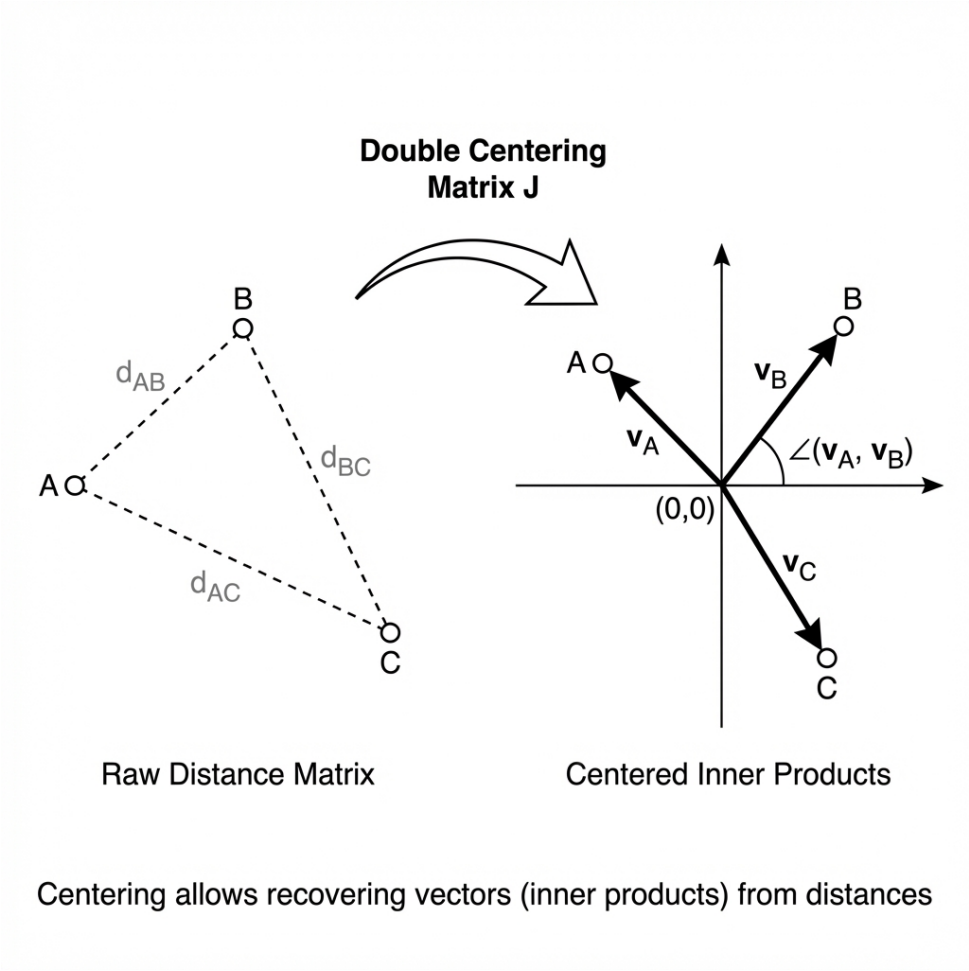


Figure 55: Visualizing Double Centering: Converting uncentered pairwise distances (left) into centered inner products (right).

3. **Eigendecomposition**: Compute the SVD (or eigendecomposition) of $\mathbf{B} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$.

4. **Embedding**: The new coordinates $\mathbf{X}_{low}$ in k dimensions are given by the top k eigenvectors scaled by the square root of their eigenvalues:

$$\mathbf{X}_{low} = \mathbf{U}_k \mathbf{\Sigma}_k^{1/2} \tag{23}$$

   Connection to Theory: This algebraic method (Classical MDS) provides the optimal solution that minimizes the *strain* (a squared-error variant of the stress function defined in Section 6.1). It constructs an **embedded space** where the pairwise Euclidean distances match the original distances δ_{ij} as closely as possible, preserving the global geometry.

5. **Result**: The 2D embedding produced by MDS closely matches the actual geographical map of the US (up to rotation and reflection), proving that the intrinsic geometry was preserved.

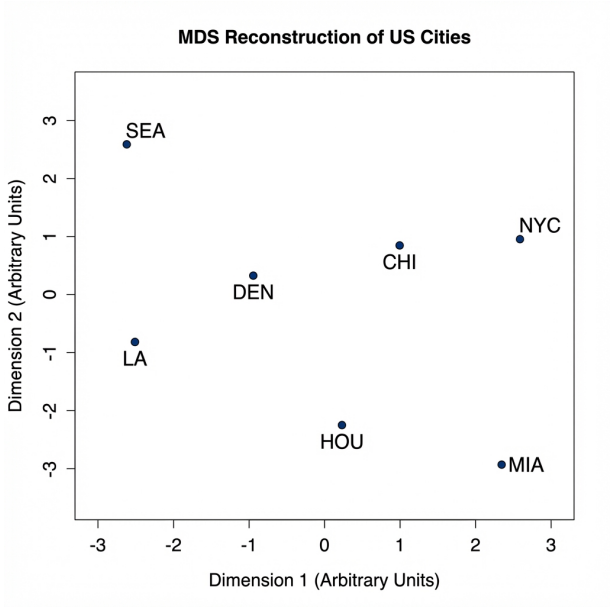


Figure 56: MDS Reconstruction of US Cities from pairwise distances.

35 Conclusion

In this lecture, we explored techniques to tame the Curse of Dimensionality. We started with **SVD** as a fundamental algebraic tool, which powers **PCA**, the standard for linear dimensionality reduction (variance maximization). We then moved to non-linear methods like **MDS**, **Isomap**, and **LLE** handling complex manifolds, and briefly touched upon **Feature Selection** for interpretability.

Lecture 6: Decision Trees and Ensemble Methods

Contents

36 Introduction

This document serves as a comprehensive study resource for Lecture 6, dedicated to **Unsupervised Learning**, with a specific focus on **Clustering**.

In the previous lecture, we explored dimensionality reduction techniques such as PCA, MDS, and Isomap. While those methods aim to simplify data representation, clustering aims to discover inherent structures by grouping similar data points together.

The lecture covers:

1. **Fundamentals of Unsupervised Learning**: The distinction between supervised and unsupervised paradigms.
2. **Clustering Basics**: Definitions, goals, and evaluation challenges.
3. **K-Means Clustering**: The algorithm, object function derivation, initialization strategies (K-Means++), and selection of K.
4. **Spectral Clustering**: A graph-theoretic approach to handling non-convex clusters.

37 Unsupervised Learning

37.1 Supervised vs. Unsupervised

Machine Learning tasks are broadly categorized based on the nature of the data:

- **Supervised Learning**: The dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$ contains input-output pairs. The goal is to learn a mapping $f : \mathcal{X} \to \mathcal{Y}$ that generalizes to unseen data. Common tasks include Classification (discrete y) and Regression (continuous y).
- **Unsupervised Learning**: The dataset $\mathcal{D} = \{\mathbf{x}_i\}$ consists strictly of input data WITHOUT labels. The goal is to infer the properties of the probability density $p(\mathbf{x})$ or to discover hidden structures within the data. Main tasks include:
  - **Clustering**: Grouping data into cohesive subsets.
  - **Dimensionality Reduction**: Finding a low-dimensional manifold that preserves data properties.
  - **Density Estimation**: Modeling the distribution $p(\mathbf{x})$.

38 Clustering Fundamentals

38.1 What is Clustering?

Clustering is the process of partitioning a set of data objects (or observations) into subsets (clusters) such that:

- Objects within a cluster are **similar** to one another (High intra-class similarity).
- Objects in different clusters are **dissimilar** (Low inter-class similarity).

Unlike classification, the "classes" are unknown beforehand. We let the data speak for itself.

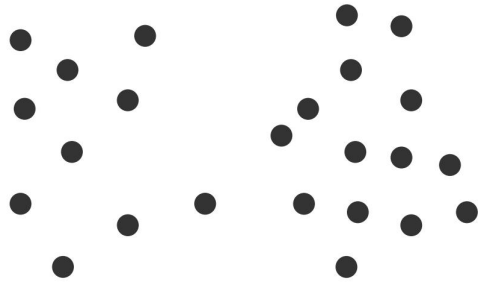


Figure 57: The goal of clustering is to identify groups (clusters) in unlabeled data.

38.2 Applications

Clustering is ubiquitous in science and industry:

- **Biology**: Finding subtypes of diseases or protein families.
- **Computer Vision**: Image segmentation or categorization.
- **Marketing**: Customer segmentation for targeted advertising.
- **Social Network Analysis**: Detecting communities or user groups.

38.3 Evaluation Methodology

Evaluating clustering is inherently difficult because there is often no "ground truth".

38.3.1 Ambiguity

Clustering is subjective. Values such as similarity are context-dependent. For example, given a set of Simpson's characters, one could cluster them by:

- **Family**: Simpsons vs. Flanders.
- **Gender**: Male vs. Female.
- **Occupation**: School employees vs. Power Plant workers.

(a) Clustered by Family

(b) Clustered by Job

(c) Clustered by Gender

Figure 58: Clustering is subjective: different criteria yield different valid partitions.

38.3.2 Quality Criteria

Despite the subjectivity, we define quality based on:

1. **Compactness (Cohesion)**: Points within a cluster should be close.
2. **Separation**: Clusters should be far apart.
3. **Stability**: The result should be robust to noise or small data perturbations.
4. **Interpretability**: The clusters should make sense in the domain context.

39 K-Means Clustering

K-Means is a **partitional** clustering algorithm. It attempts to split the dataset of N points into K disjoint subsets $C_1, \ldots, C_K$ to minimize the within-cluster sum of squares.

39.1 The Algorithm (Lloyd's Algorithm)

The algorithm is an iterative process that converges to a local minimum.

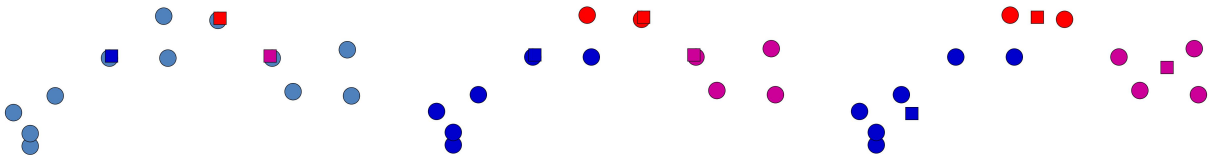
1. **Initialization**: Select K random points as initial centroids $\mu_1, \ldots, \mu_K$.
2. **Assignment Step**: Assign each data point $\mathbf{x}_i$ to the closest centroid.

$$C_k = \{\mathbf{x}_i : \|\mathbf{x}_i - \mu_k\|^2 \leq \|\mathbf{x}_i - \mu_j\|^2, \forall j \neq k\}$$

3. **Update Step**: Recompute the centroid of each cluster as the mean of its assigned points.

$$\mu_k = \frac{1}{|C_k|} \sum_{\mathbf{x} \in C_k} \mathbf{x}$$

4. **Repeat**: Iterate steps 2 and 3 until the centroids do not change (convergence).



(a) Initialization (b) Assignment Step (c) Update Step

Figure 59: K-Means Algorithm steps: (a) Random initialization of centroids (x), (b) Points assigned to nearest centroid, (c) Centroids updated to the mean of the cluster.

39.2 Objective Function and Convergence

K-Means strictly minimizes the **Within-Cluster Sum of Squares (WCSS)**, also known as inertia.

Let $r_{nk} \in \{0, 1\}$ be an indicator variable such that $r_{nk} = 1$ if data point $\mathbf{x}_n$ belongs to cluster k, and 0 otherwise. The objective function J is:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \mu_k\|^2 \tag{24}$$

This is a non-convex function, meaning there are many local minima.

- The **Assignment Step** minimizes J with respect to r_{nk} (keeping μ_k fixed). It assigns $\mathbf{x}_n$ to the μ_k that gives the smallest squared distance.
- The **Update Step** minimizes J with respect to μ_k (keeping r_{nk} fixed). Differentiating J w.r.t μ_k and setting to 0 gives the mean.

Since J decreases strictly (or stays same) at each step and is bounded below by 0, the algorithm is guaranteed to converge, though not necessarily to the global optimum.

39.3 Voronoi Tessellation

The assignment rule implies that K-Means partitions the space into **Voronoi cells**. The decision boundaries between clusters are linear bisectors of the segments connecting the centroids. This implies that K-Means effectively finds **convex, spherical clusters**.

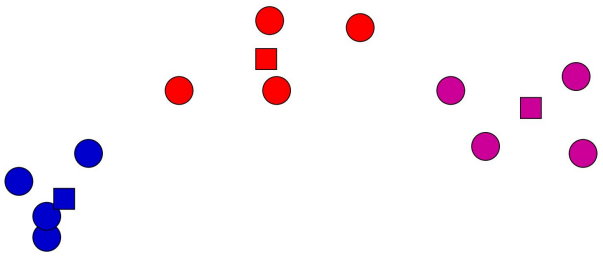
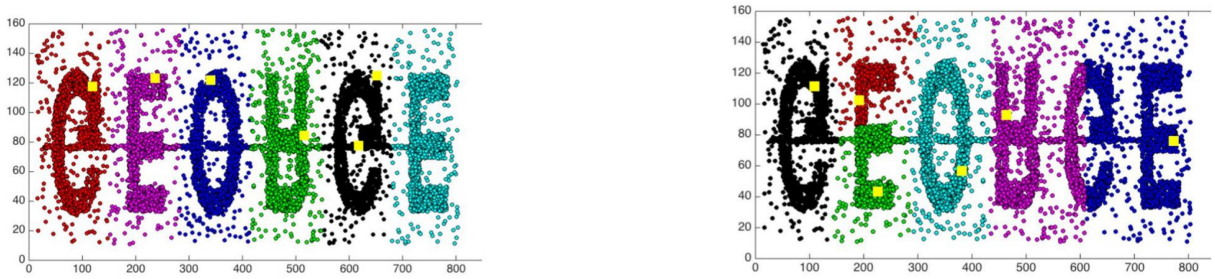


Figure 60: K-Means partitions the space into convex Voronoi cells.

39.4 Initialization and K-Means++

Because K-Means is sensitive to initialization, a poor choice of starting centroids can lead to a suboptimal clustering (bad local minimum).



(a) Local Minimum 1

(b) Local Minimum 2

Figure 61: The final result depends heavily on the initial seeds (Instability).

39.4.1 K-Means++ Initialization

Standard K-Means initialization (randomly selecting K points) is risky because the chosen points might be clustered together in a single dense region, leading to a poor local minimum. **K-Means++** is a smart initialization strategy that spreads out the initial centroids.

The algorithm works as follows:

1. **First Centroid**: Choose the first centroid μ_1 uniformly at random from the dataset.
2. **Distance Computation**: For every data point $\mathbf{x}_i$, compute $D(\mathbf{x}_i)$, the distance to the *nearest* centroid that has already been chosen.
3. **Probabilistic Selection**: Choose the next centroid μ_{next} from the data points, where the probability of choosing a point $\mathbf{x}$ is proportional to its squared distance $D(\mathbf{x})^2$:

$$P(\mu_{next} = \mathbf{x}) = \frac{D(\mathbf{x})^2}{\sum_{\mathbf{x}' \in \mathcal{D}} D(\mathbf{x}')^2} \tag{25}$$

   Intuition: Points that are far away from existing centroids have a high $D(\mathbf{x})^2$ and thus a high probability of being selected. This forces the algorithm to pick the next centroid in an "uncovered" part of the space.
4. **Repeat**: Repeat steps 2 and 3 until all K centroids are chosen. This is a finite process that stops exactly when we have K centers.
5. **Run K-Means**: Proceed with standard Lloyd's algorithm using these K centroids as the starting configuration.

Why Probabilistic? You might ask: why not just pick the *furthest* point deterministically? A deterministic "furthest point" strategy is extremely sensitive to **outliers**. If an outlier exists far from the data, it would definitely be chosen as a centroid, wasting a cluster. By using probability proportional to distance, K-Means++ balances **separation** with the need to pick points from **dense regions**, ensuring robust and well-separated centroids.

39.5 Choosing K

The number of clusters K is a hyperparameter. A common heuristic is the **Elbow Method**:

1. Run K-Means for a range of K values (e.g., 1 to 10).
2. Calculate the WCSS (Distortion) for each K.
3. Plot WCSS vs. K.
4. Choose the K at the "elbow" of the curve, where adding more clusters gives diminishing returns in variance reduction.

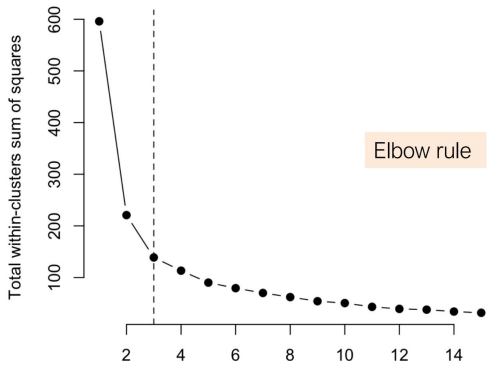
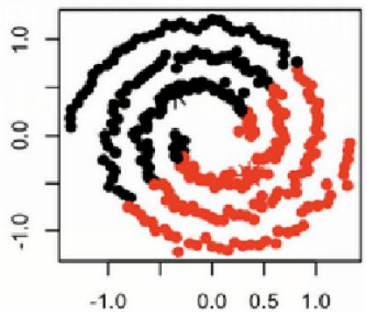


Figure 62: The Elbow Method suggests $K = 3$ here.

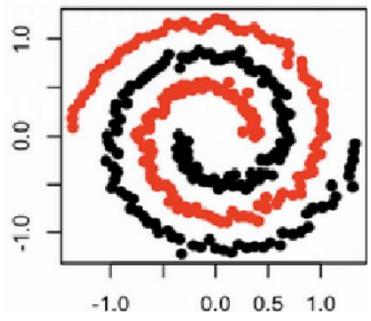
40 Spectral Clustering

40.1 Motivation: Non-Convex Clusters

K-Means fails when clusters have non-convex shapes (e.g., concentric circles, spirals) because it relies on Euclidean distance from a center.



(a) K-Means Failure: It cuts through the rings linearly.



(b) Spectral Clustering Success: It captures the connectivity.

Figure 63: Comparison on non-convex data.

40.2 Graph-Based Perspective

Spectral clustering views data points as nodes in a graph $G = (V, E)$. The goal is to partition the graph into subgraphs such that edges within subgraphs have high weights (similarity) and edges between subgraphs have low weights (cut).

40.2.1 Similarity Graph Construction

First, we build a similarity graph where edge weights W_{ij} represent similarity between $\mathbf{x}_i$ and $\mathbf{x}_j$.

- **ϵ-neighborhood graph**: Connect points if $\|\mathbf{x}_i - \mathbf{x}_j\| < \epsilon$.
- **k-Nearest Neighbors graph**: Connect node i to j if j is among the k-nearest neighbors of i.
- **Fully connected graph**: Connect all points with Gaussian similarity weights:

$$W_{ij} = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right)$$

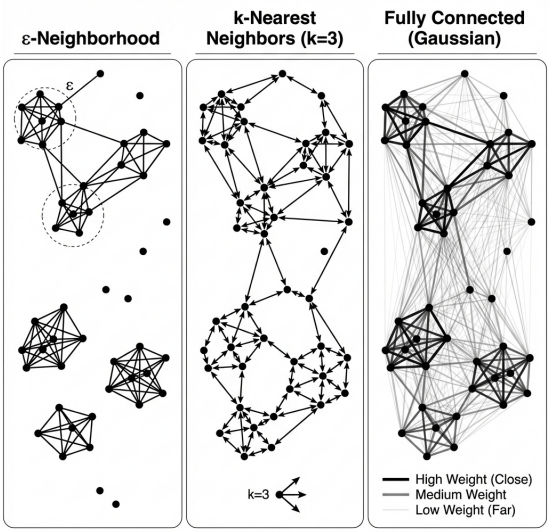


Figure 64: Three ways to construct a similarity graph: ϵ-neighborhood, k-Nearest Neighbors, and Fully Connected.

40.3 The Graph Laplacian

To interpret the graph structure algebraically, we define:

- **Adjacency Matrix** W: The $N \times N$ matrix of weights.
- **Degree Matrix** D: A diagonal matrix where $D_{ii} = \sum_j W_{ij}$ (sum of connections).

The **Unnormalized Graph Laplacian** is defined as:

$$L = D - W \tag{26}$$

40.3.1 Properties of the Laplacian

The Laplacian has a crucial property: For any vector $\mathbf{f} \in \mathbb{R}^N$ (which we can think of as assigning a value f_i to each node i):

$$\mathbf{f}^T L \mathbf{f} = \mathbf{f}^T (D - W)\mathbf{f} = \sum_{i=1}^{N} D_{ii} f_i^2 - \sum_{i,j}^{N} f_i W_{ij} f_j = \frac{1}{2} \sum_{i,j=1}^{N} W_{ij} (f_i - f_j)^2 \tag{27}$$

This quadratic form measures the **"smoothness"** of the signal $\mathbf{f}$ over the graph:

- The term $(f_i - f_j)^2$ measures the difference in values between nodes i and j.
- The weight W_{ij} ensures that this difference is penalized *only if* nodes i and j are strongly connected (similar).
- Therefore, minimizing $\mathbf{f}^T L \mathbf{f}$ finds a vector $\mathbf{f}$ that changes very little between similar data points ($f_i \approx f_j$ when W_{ij} is large).

This connects directly to clustering: a good cluster indicator vector should be piecewise constant (smooth) within clusters but can change abruptly between clusters (where W_{ij} is small). This explains why we seek vectors $\mathbf{f}$ that minimize $\mathbf{f}^T L \mathbf{f}$.

Connection to Eigenvalues: We want to find $\mathbf{f}$ that minimizes this smoothness energy. To avoid the trivial solution $\mathbf{f} = \mathbf{0}$, we impose a constraint $\|\mathbf{f}\|^2 = 1$. This leads to the optimization problem:

$$\min_{\mathbf{f}} \mathbf{f}^T L \mathbf{f} \quad \text{subject to} \quad \|\mathbf{f}\| = 1 \tag{28}$$

Using Lagrange multipliers, the solution to this problem satisfies $L\mathbf{f} = \lambda \mathbf{f}$. Thus, the optimal vectors are the **eigenvectors** of L, and the minimum value of the objective function is the corresponding **eigenvalue** λ:

$$\mathbf{f}^T L \mathbf{f} = \mathbf{f}^T (\lambda \mathbf{f}) = \lambda \|\mathbf{f}\|^2 = \lambda \tag{29}$$

Therefore, the eigenvectors associated with the *smallest* eigenvalues provide the "smoothest" possible functions over the graph, which effectively act as soft indicators for the clusters.

40.4 The Spectral Clustering Algorithm

1. **Graph Construction**: Construct the similarity graph and its adjacency matrix W.
2. **Laplacian**: Compute the unnormalized Laplacian $L = D - W$ (or normalized versions $L_{sym} = I - D^{-1/2} W D^{-1/2}$).
3. **Eigendecomposition**: Compute the first k eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_k$ corresponding to the k smallest eigenvalues of L.
4. **Projection**: Form the matrix $U \in \mathbb{R}^{N \times k}$ containing these eigenvectors as columns. *Note:* The rows of U, denote them $\mathbf{y}_i \in \mathbb{R}^k$, are the new representations of the data points $\mathbf{x}_i$.
5. **Clustering**: Cluster the points $\{\mathbf{y}_i\}_{i=1}^N$ using K-Means into k clusters.

40.4.1 Why it works

Mapping points to the eigenvector space (U) reveals the cluster structure. Consider a graph with k ideally disjoint connected components. In this case, the matrix L is block-diagonal, and the eigenvalue 0 has multiplicity k. The corresponding eigenvectors act as strict **indicator vectors**: for an eigenvector associated with a specific cluster, the elements corresponding to points in that cluster are **non-zero** (and constant), while elements for all other points are **zero**.

In real-world scenarios where the graph is connected but correctly clustered (weakly connected components), the smallest eigenvalues are not exactly 0 but are very close to it. Consequently, the eigenvectors serve as a **"soft" indication** of cluster membership. Points within the same cluster will have eigenvector values clustered around a specific constant, distinct from the values of points in other clusters. This separation makes it easy for the final K-Means step to group the points in the eigenvector space.

41 Summary: K-Means vs. Spectral

Feature	**K-Means**	**Spectral Clustering**
Cluster Shape	Convex (Spherical/Voronoi)	Arbitrary (can handle rings, spirals)
Input	Raw coordinates $\mathbf{x}$	Similarity Matrix W (Graph)
Parameters	K	K, σ (kernel), Neighbor definition
Complexity	$O(N \cdot K \cdot d \cdot \text{iters})$ (Linear)	$O(N^3)$ (Eigendecomposition)
Scalability	High (suitable for large N)	Poor (expensive for large N)

Table 2: Comparison of Clustering Methods

42 Lab Session: Clustering Implementation

In this lab, we implemented and compared K-Means and Spectral Clustering on both synthetic and real-world datasets.

42.1 K-Means on Synthetic Data

We applied K-Means to a synthetic dataset with 4 distinct clusters.

- **Goal**: Observe the algorithm's convergence and partition.
- **Result**: As expected for well-separated convex clusters, K-Means successfully identifies the 4 groups.

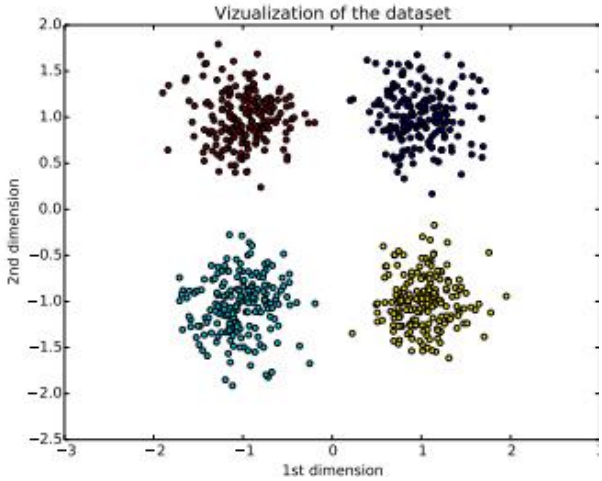


Figure 65: K-Means correctly clustering a synthetic dataset with $K = 4$.

42.2 K-Means on MNIST (Reduced)

We clustered a subset of the MNIST handwritten digits dataset (1000 samples), reduced to 8 dimensions via PCA.

- **Elbow Method**: By plotting WCSS vs. K, we observed an "elbow" around $K = 10$, consistent with the 10 distinct digits (0-9).
- **Outcome**: Visualizing the clusters (in 2D PCA space) shows that K-Means groups similar digits, though some overlap exists due to writing styles.

42.3 Spectral Clustering vs. K-Means on Non-Convex Data

We tested both algorithms on a "two spirals" or "nested rings" dataset to highlight their differences.

1. **K-Means Failure**: K-Means assumes spherical clusters. On non-convex shapes, it partitions the space based on Euclidean distance from a center, essentially "cutting" through the natural manifold structure.

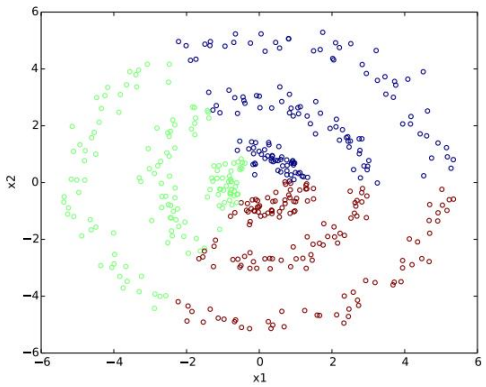


Figure 66: K-Means fails on non-convex data, creating linear boundaries.

2. **Spectral Clustering Success**: By constructing a k-Nearest Neighbors graph (capturing local connectivity) and clustering the eigenvectors of the Laplacian, Spectral Clustering correctly identifies the two separate manifolds.

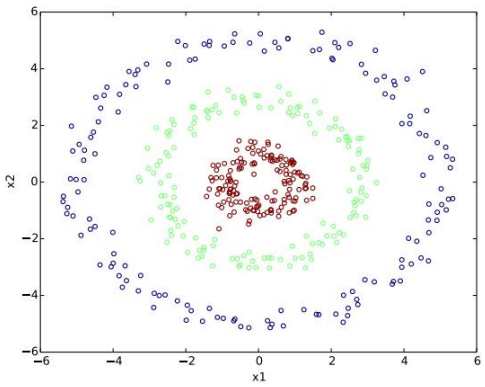


Figure 67: Spectral Clustering successfully unrolls the non-convex structure.

Lecture 7: Dimensionality Reduction

Contents

43 Introduction

This lecture introduces **Non-parametric Learning**, a paradigm distinct from the parametric methods (like Linear and Logistic Regression) discussed in previous sessions. We specifically focus on **Nearest Neighbour Methods** (K-NN), which are simple yet powerful algorithms for both classification and regression.

43.1 Parametric vs. Non-Parametric Learning

To understand non-parametric learning, we first contrast it with the parametric approach.

43.1.1 Parametric Models

In parametric learning (e.g., Linear Regression, Logistic Regression), we assume a functional form for the relationship between the input $\mathbf{x}$ and the output y.

- **Assumption**: The data follows a specific distribution or model (e.g., $y = \mathbf{w}^T\mathbf{x} + b$).
- **Learning**: Involves estimating a fixed set of parameters $\boldsymbol{\theta}$ from the training data.
- **Complexity**: The number of parameters is fixed and independent of the training set size size.
- **Pros**: Simple, fast prediction, interpretable, effective with limited data.
- **Cons**: Limited expressiveness. If the assumption is wrong (mismodeling), the model performs poorly (high bias).

43.1.2 Non-Parametric Models

Non-parametric methods make fewer assumptions about the underlying data distribution.

- **Assumption**: The data speaks for itself; we do not assume a fixed global model structure (like linearity).
- **Complexity**: The complexity of the model grows with the number of training examples. Effectively, the number of parameters is infinite or proportional to the dataset size N.
- **Pros**: Flexible, capable of modeling complex, non-linear relationships, robust to mismodeling.
- **Cons**: Requires large amounts of data, prone to overfitting if not regularized, computationally expensive (slow prediction) as they often store the entire dataset.

43.2 Instance-Based Learning

K-NN is a type of **Instance-Based Learning** (or "Lazy Learning").

1. **Learning (Lazy)**: Simply store the training examples. There is no explicit training phase or model parameter optimization.

2. **Prediction**: When a new query point arrives, the algorithm actively computes the output by comparing the query to the stored instances.

This is analogous to "Case-Based Reasoning" (e.g., a doctor treating a patient based on similar past cases) or legal precedents.

44 Similarity and Distance Measures

The core of nearest neighbour methods is the notion of **similarity** or **distance**. Determining which examples are "close" to each other depends heavily on the chosen metric and the nature of the data features.

44.1 Continuous Features

For data vectors $\mathbf{x}, \mathbf{z} \in \mathbb{R}^n$, we typically use L_p norms (Minkowski distance).

44.1.1 Euclidean Distance (L_2 Norm)

The spectral distance, representing the length of the straight line segment between two points.

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|_2 = \sqrt{\sum_{j=1}^{n} (x_j - z_j)^2} \tag{30}$$

44.1.2 Manhattan Distance (L_1 Norm)

The sum of absolute differences. Also known as "Taxi-cab" geometry, as it represents the distance traveled along a grid.

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|_1 = \sum_{j=1}^{n} |x_j - z_j| \tag{31}$$

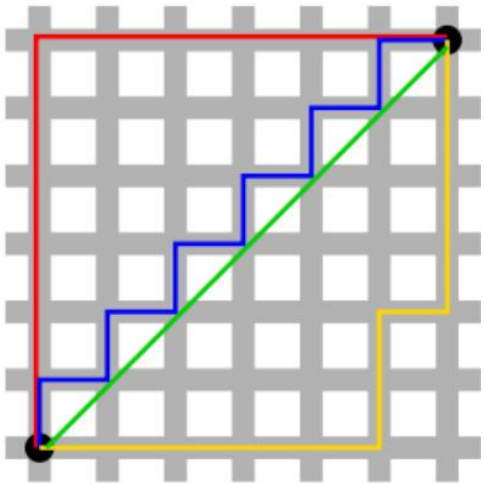


Figure 68: Manhattan (Red/Blue/Yellow) vs. Euclidean (Green) distance.

44.1.3 Correlation and Cosine Similarity

Sometimes the magnitude of the vectors matters less than their orientation or profile.

Pearson's Correlation: Measures the linear correlation between two variables (or vectors). For centered data ($\bar{x} = 0, \bar{z} = 0$):

$$\rho(\mathbf{x}, \mathbf{z}) = \frac{\sum x_j z_j}{\sqrt{\sum x_j^2}\sqrt{\sum z_j^2}} = \frac{\langle \mathbf{x}, \mathbf{z} \rangle}{\|\mathbf{x}\|\|\mathbf{z}\|} = \cos(\theta) \tag{32}$$

Cosine Similarity: Defined as the cosine of the angle θ between two vectors. It ranges from -1 (opposite) to 1 (identical direction).

- **Scale Invariant**: Two vectors $\mathbf{x}$ and $\alpha\mathbf{x}$ ($\alpha > 0$) have similarity 1.
- **Application**: Text mining. Two documents are similar if they share the same word distribution, regardless of document length (frequency magnitude).

44.2 Categorical and Binary Features

When features indicate the presence or absence of a property (e.g., bits '0' or '1'), Euclidean distance is rarely optimal.

44.2.1 Hamming Distance

The number of positions at which the corresponding symbols generally differ. For binary strings, this is the count of XOR operations:

$$d(\mathbf{x}, \mathbf{z}) = \sum_{j=1}^{n} (x_j \oplus z_j) \tag{33}$$

This is equivalent to the L_1 distance for binary vectors.

44.2.2 Jaccard Similarity

Measures the overlap between two sets (binary vectors) relative to their union. Useful when the "Check" (1s) is much more informative than the "Absence" (0s) (e.g., sparse data).

$$J(\mathbf{x}, \mathbf{z}) = \frac{|\mathbf{x} \cap \mathbf{z}|}{|\mathbf{x} \cup \mathbf{z}|} = \frac{\sum (x_j \wedge z_j)}{\sum (x_j \vee z_j)} \tag{34}$$

Example 1. *Consider* $\mathbf{x} = 010101001$ *and* $\mathbf{y} = 010011000$.

- ***Hamming****: Mismatches at indices 3, 4, 8.* $d = 3$.
- ***Jaccard****: Intersection (1s in both) = 2 positions. Union (1s in either) = 5 positions.* $J = 2/5 = 0.4$.

45 The K-Nearest Neighbours (K-NN) Algorithm

45.1 Algorithm Definition

Given a training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$ and a new query point $\mathbf{x}_q$:

Algorithm 1 K-Nearest Neighbours Prediction

Input: Training data $\mathcal{D}$, Query $\mathbf{x}_q$, Hyperparameter K.
Compute $d(\mathbf{x}_q, \mathbf{x}^{(i)})$ for all $i = 1 \dots N$.
Sort the distances and select the set $\mathcal{N}_K$ of the K indices with the smallest distances.
if Classification (Discrete y) **then**
    Return the majority class (mode) of $\{y^{(j)} : j \in \mathcal{N}_K\}$.
else if Regression (Continuous y) **then**
    Return the average value: $\hat{y} = \frac{1}{K} \sum_{j \in \mathcal{N}_K} y^{(j)}$.
end if

45.2 Effect of K

The hyperparameter K controls the model complexity (bias-variance trade-off).

- **Small K (e.g., K=1)**:
  - Highly flexible decision boundary.
  - Captures local structure but also captures **noise**.
  - **Overfitting** (Low Bias, High Variance).
- **Large K**:
  - Smoother decision boundary.
  - Suppresses noise by averaging, but may smooth out distinct local patterns.
  - If $K = N$, the prediction is just the global average/majority.
  - **Underfitting** (High Bias, Low Variance).

45.3 Choosing K

Since K governs the complexity, it should be selected via **Cross-Validation**.

1. Split training data into folds.
2. For each candidate K (e.g., $1, 3, 5, \dots$), measure the validation error.
3. Choose the K with the lowest validation error.

Rule of Thumb: A common starting point is $K \approx \sqrt{N}$.

45.4 Decision Boundaries and Voronoi Tessellations

For 1-NN, the decision boundary is defined by the **Voronoi Tessellation**. The space is partitioned into cells, where each cell V_i contains all points closer to training example $\mathbf{x}^{(i)}$ than to any other.

- The boundary between two cells is the perpendicular bisector of the segment connecting the two points.
- The decision boundary is the union of the boundaries of the cells belonging to different classes.

46 Computational Efficiency and K-D Trees

46.1 The Problem: Distance Calculation Cost

The standard K-NN algorithm is computationally expensive. To classify a single new query point $\mathbf{x}_q$, we must calculate the distance between $\mathbf{x}_q$ and **every single point** in the training set $(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)})$ to find the smallest ones.

- **Naive Complexity**: $O(N \cdot D)$, where N is the number of samples and D is the number of dimensions.
- **Issue**: If we have $N = 1,000,000$ images, we perform 1 million distance checks for *one* prediction. This is too slow for real-time applications.

Goal: Can we structure the data so that we know certain points are "too far away" without actually calculating the distance?

46.2 Intuition: The "Binary Search" of Multidimensional Space

Think of searching for a word in a dictionary. You don't read every word from the first page. You open the middle. If your word is alphabetically after the middle word, you ignore the entire first half of the book. **K-D Trees (k-Dimensional Trees)** extend this logic to multiple dimensions:

- In 1D, we split points left/right of a median number.
- In 2D, we split points left/right of a vertical line, then above/below horizontal lines.

46.3 K-D Tree Construction (Building the Index)

The K-D tree is a binary tree that recursively partitions the space into smaller "boxes" (hyper-rectangles).

1. **Choose Axis**: Start with axis x_1 (dimension 0).
2. **Find Median**: Sort the dataset along this axis and find the median point.
3. **Split**:
    - **Left Child**: All points with $x_1 <$ median.

- **Right Child**: All points with $x_1 \geq$ median.

4. **Recurse**: Move to the next axis x_2 (dimension 1) and repeat the process for both the Left and Right children. Cycle through axes ($x_1 \to x_2 \to \cdots \to x_D \to x_1$).

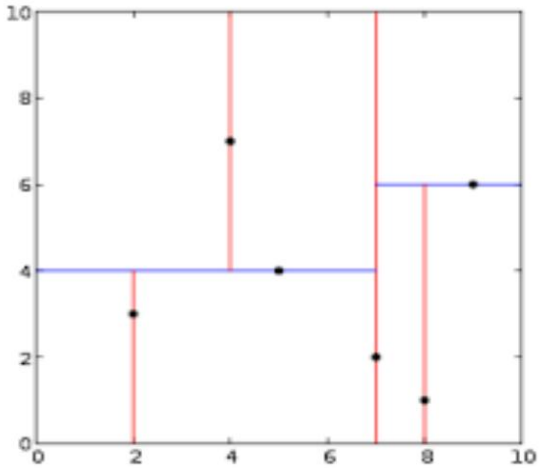


Figure 69: Visualizing Construction: The first cut (vertical) splits the whole space. The second cuts (horizontal) split the sub-regions.

46.4 K-NN Search with Pruning

How does this structure help us search?

- **Idea**: We maintain a "Current Best" nearest neighbour found so far, at distance R (radius).
- **Pruning Rule**: If the entire geometric box of a tree branch is further than R from our query, we simply **skip** that whole branch. We don't need to check any points inside it.

46.4.1 Algorithm Steps

Given a query $\mathbf{x}_q$:

1. **Descend**: Start at the root. Walk down the tree (comparing coordinates) just like inserting a new point, until you hit a Leaf node.
2. **Initial Guess**: Calculate the distance to the point in the leaf. This is our "Current Best", with distance R.
3. **Backtrack (The Check)**: Walk back up the tree to the parent node. The parent represents a splitting plane (e.g., $x_1 = 5$).
   - **Check Intersection**: Does a ball of radius R around $\mathbf{x}_q$ cross the splitting plane?
   - **If NO (Prune)**: The "other side" of the plane is strictly further away than our current best. Ignore the other child. (Efficiency Gain!)
   - **If YES**: There *could* be a closer point on the other side. We must "open" that branch and search it.

46.4.2 Complexity

- **Average Case**: $O(\log N)$. We prune many branches, making it very fast ('logarithmic' time).
- **Worst Case**: $O(N)$. If the points are high-dimensional, the "ball" often intersects all boundaries, and we search everything.

46.5 The Curse of Dimensionality

K-D Trees work well for low dimensions ($D < 20$). As D increases:

- **Data Sparsity**: Points become sparsely scattered.
- **Distance Convergence**: The distance to the nearest neighbour and the furthest neighbour becomes similar.
- **Ineffective Pruning**: In high dimensions, the "Ball" of radius R intersects almost all splitting planes. We end up visiting all nodes, reverting to $O(N)$.

47 Variants and Practical Considerations

47.1 Feature Normalization

K-NN is highly sensitive to the scale of features. If feature x_1 ranges from 0-1 and x_2 ranges from 0-1000, x_2 will dominate the Euclidean distance. **Standardization** (Z-score normalization) is essential:

$$x_{new} = \frac{x - \mu}{\sigma} \tag{35}$$

47.2 Weighted K-NN

Instead of a simple majority vote, we can weight the votes by the inverse of the distance. Closer neighbours effectively "shout louder".

$$w_i = \frac{1}{d(\mathbf{x}_q, \mathbf{x}^{(i)})^2 + \epsilon} \tag{36}$$

This allows us to set a larger K (to reduce variance) without letting distant points overly influence the local decision.

48 Lab Session: Handwritten Digit Recognition

In the practical session, we applied the K-NN classifier to a real-world problem: Optical Character Recognition (OCR) for handwritten digits using the **MNIST** dataset.

48.1 Dataset Overview

The MNIST database contains grayscale images of handwritten digits $(0 - 9)$.

- **Resolution**: Each image is 28×28 pixels.
- **Features**: We flatten the image into a vector of 784 features ($x \in [0, 255]$).
- **Size**: The full dataset has 60,000 training samples and 10,000 testing samples.

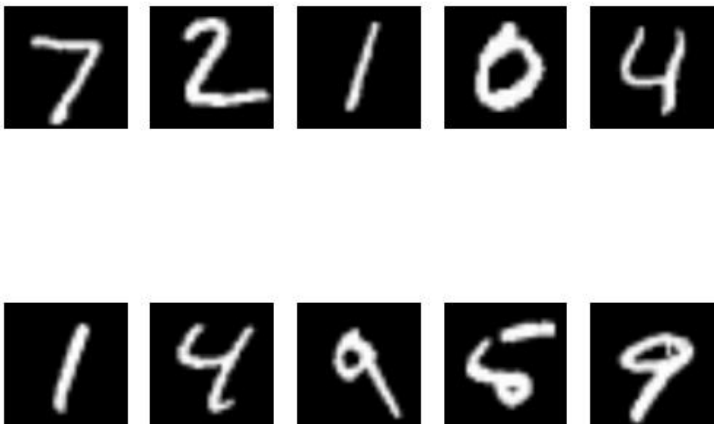


Figure 70: Samples from the MNIST training set.

48.2 Implementation Details

We implemented the K-NN algorithm from scratch (without *scikit-learn*).

1. **Loading**: We loaded the data and normalized it (or used raw pixel values).
2. **Distance**: We used Euclidean distance.
3. **Prediction**: For each test image, we found the k nearest training images.

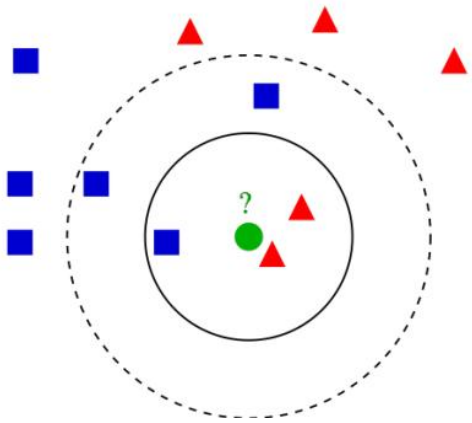


Figure 71: Conceptual visualization of K-NN classification with $k = 3$ vs $k = 5$.

48.3 Observations

- **Accuracy**: K-NN performs surprisingly well on this task (often $> 90\%$ accuracy) despite its simplicity.
- **Performance**: The main bottleneck is the **prediction time**. Classifying a single image requires comparing it 60,000 times. In the lab, we restricted the training set to a smaller subset (e.g., 2,000 images) to keep the runtime manageable.
- **Hyperparameters**: We observed that $k = 1$ provides very flexible boundaries but is sensitive to outliers, while larger k values smooth the decision boundary.

Lecture 8: Clustering

Contents

49 Introduction

In the previous lecture, we explored **Non-Parametric Learning**, specifically the K-Nearest Neighbours (K-NN) algorithm. As a reminder, non-parametric methods do not assume a fixed model structure (like a linear equation) but instead let the data dictate the decision boundary. The "complexity" of the model effectively grows with the number of training points.

While K-NN is simple and powerful, it suffers from significant drawbacks:

- **Memory Intensive**: It requires storing the entire training dataset.
- **Prediction Latency**: To classify a single new point, we naively compute its distance to *all* N training examples, resulting in $O(N \cdot D)$ complexity.

This lecture addresses these challenges and introduces more advanced non-parametric methods. We begin by optimizing K-NN search using **K-D Trees**, then move to a new class of algorithms: **Decision Trees**. Finally, we explore **Ensemble Learning**, a strategy to combine multiple models (like trees) to boost performance, specifically covering **Bagging** (Random Forests) and **Boosting** (AdaBoost).

50 Accelerating K-NN: K-D Trees

To make K-NN practical for large datasets, we need to avoid the brute-force distance calculation ($O(N)$). The goal is to organize the data spatially so we can quickly prune away vast regions of the search space that are definitely too far away.

50.1 Concept: Multidimensional Binary Search

The K-Dimensional Tree (K-D Tree) is a generalization of the binary search tree (BST) to multi-dimensional space.

- In a 1D BST, we compare a value to the current node and go left (smaller) or right (larger).
- In a K-D Tree, we cycle through the dimensions (axes) at each level of the tree.
  - *Level 0*: Split based on dimension x_1.
  - *Level 1*: Split based on dimension x_2.
  - ...
  - *Level* $D-1$: Split based on dimension x_D.
  - *Level* D: Cycle back to x_1.

Each node essentially defines a splitting hyperplane perpendicular to one of the coordinate axes, dividing the space into two half-spaces.

50.2 Tree Construction Algorithm

To build a balanced K-D tree, we recursively split the data at the **median** of the chosen dimension.

Algorithm 2 Build K-D Tree

```
function BuildTree(points, depth)
    if points is empty then
        return null
    end if
    k ← dimensionality of data
    axis ← depth (mod k)                          ▷ Select axis based on depth
    Sort points by coordinates in axis
    median ← median index of points
    node ← point at median
    node.left ← BuildTree(points[0 ... median-1], depth + 1)
    node.right ← BuildTree(points[median+1 ... end], depth + 1)
    return node
end function
```

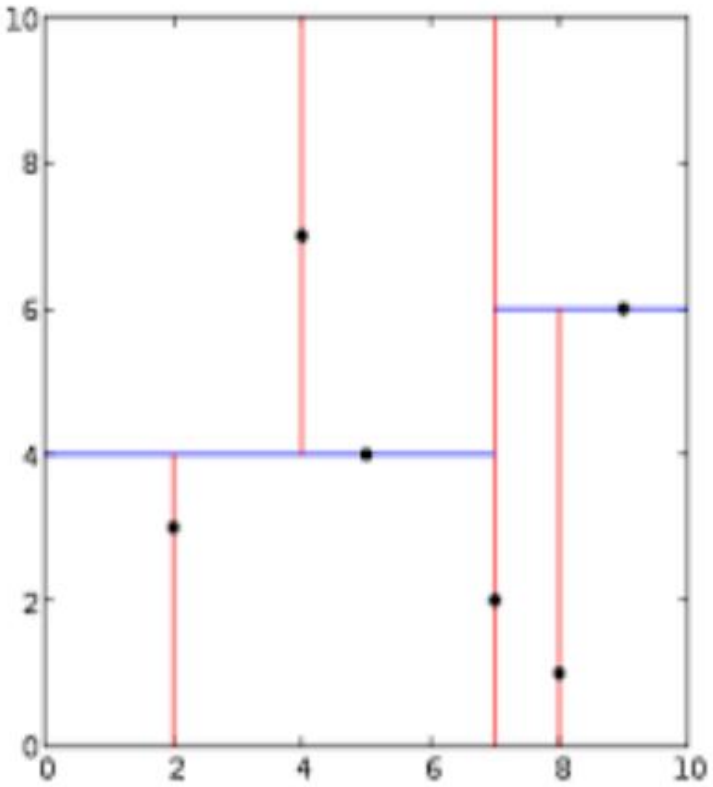


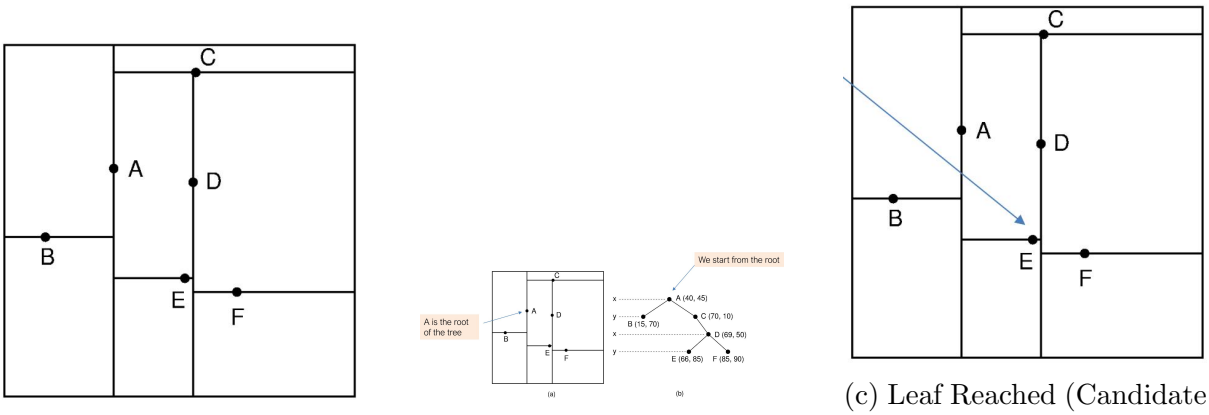
Figure 72: Construction of a 2D K-D Tree. The first split (red vertical line) divides the space based on x-coordinates. The next splits (blue horizontal lines) divide the subspaces based on y-coordinates.

50.3 Nearest Neighbour Search

Searching for the nearest neighbour of a query point **q** is efficient because we can discard entire subtrees.

Logic:

1. **Descend**: Traverse the tree from root to leaf, comparing **q**'s coordinates to the nodes. This gives us a "candidate" nearest neighbour and establish a "Current Best Distance" (R).
2. **Backtrack**: Move back up the tree. At each node, we check:
   - Does the hypersphere centered at **q** with radius R cross the splitting plane of the current node?
   - **If No**: The entire other side of the plane is too far. *Prune* that branch.
   - **If Yes**: There might be a closer point on the other side. We must investigate that branch recursively.



(a) Query Point (red cross) (b) Comparison at Root (c) Leaf Reached (Candidate E)

Figure 73: Tracing the K-D Tree search. We descend to leaf E. Then, we check if the circle around the query intersects the bounding boxes of other branches.

50.4 Complexity and the Curse of Dimensionality

- **Construction**: $O(N \log N)$ (sorting at each level).
- **Query**:
  - Average case: $O(\log N)$.
  - Worst case: $O(N)$.

K-D Trees are excellent for low dimensions ($D \lesssim 20$). However, as D increases, we encounter the **Curse of Dimensionality**. In high dimensions, points become "sparse", and Euclidean distances tend to concentrate (the ratio of the nearest to furthest distance approaches 1). Geometrically, the search hypersphere overlaps with almost all splitting planes, forcing the algorithm to visit nearly all nodes, degrading performance back to $O(N)$. For very high-dimensional data, approximate nearest neighbour methods (like Locality Sensitive Hashing) are often preferred.

51 Decision Trees

Decision Trees are a hierarchical model used for both classification and regression. Unlike K-NN, which uses the entire dataset at inference time, a decision tree "compiles" the data into a set of logical rules.

51.1 Model Structure

A decision tree consists of:

- **Nodes**: Performed a test on a specific attribute (e.g., "Is Income ¿ 50k?").
- **Branches**: Outcomes of the test (e.g., "Yes", "No").
- **Leaves**: Final class labels or values.

This structure partitions the input space into axis-aligned rectangles (decision boundaries are always perpendicular to axes).

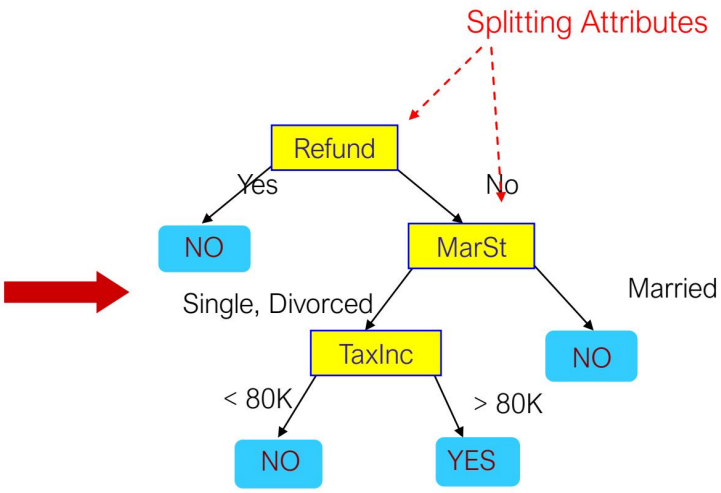


Figure 74: Example of a Decision Tree for detecting tax cheats.

51.2 Training: Hunt's Algorithm

Building a tree is a **top-down, greedy** process. The standard approach (Hunt's Algorithm) is recursive: Let D_t be the set of training records that reach node t.

1. **Base Cases (Stopping Criteria)**:
   - If all records in D_t belong to the same class y, t is a leaf node labelled y.
   - If D_t is empty, t is a leaf labelled with the default/majority class of its parent.
   - If no attributes are left to split on, allow majority voting.
2. **Recursive Step**:
   - Select the "best" attribute to split D_t.
   - Split D_t into subsets D_{child} based on the attribute's values.
   - Recursively call the procedure for each child.

51.3 Splitting Criteria

The core decision is: *Which attribute offers the best split?* We want the child nodes to be "purer" than the parent node (i.e., contain mostly one class).

51.3.1 Gini Index

The Gini Index measures the impurity of a node. Let D_t be the set of training records at node t, with total size $N_t = |D_t|$. Let $N_{t,j}$ be the number of records in D_t that belong to class j. Then, $p(j|t)$ is the **relative frequency** of class j at node t:

$$p(j|t) = \frac{N_{t,j}}{N_t} \tag{37}$$

The Gini Index for the node is defined as:

$$\text{Gini}(t) = 1 - \sum_{j=1}^{J} p(j|t)^2 \tag{38}$$

- **Minimum (0.0)**: All records belong to the same class (Pure).
- **Maximum** $(1 - 1/J)$: Records are equally distributed among all classes (Impure).

To evaluate a split on attribute A into k children $(v_1, \ldots, v_k)$, we calculate the weighted average impurity:

$$\text{Gini}_{\text{split}}(A) = \sum_{i=1}^{k} \frac{n_i}{N} \text{Gini}(child_i) \tag{39}$$

where n_i is the number of records in child i and N is the total records at the parent. We choose the attribute that **minimizes** this weighted Gini index (or maximizes the impurity reduction Δ).

51.3.2 Entropy and Information Gain

Alternatively, we can use Entropy (from Information Theory):

$$\text{Entropy}(t) = -\sum_{j=1}^{J} p(j|t) \log_2 p(j|t) \tag{40}$$

The improvement is called **Information Gain**:

$$\text{Gain}(A) = \text{Entropy}(parent) - \text{Entropy}_{\text{split}}(A) \tag{41}$$

We choose the attribute that **maximizes** Information Gain.

Intuition and Derivation

Why use $-\log_2 p$? This comes from Shannon's Information Theory. We want a measure of "uncertainty" or "surprise" associated with a random variable Y (the class label).

1. **Self-Information**: Consider an event y occurring with probability $p(y)$.

- If $p(y) = 1$ (certainty), the event carries no information (0 bits).
- If $p(y)$ is low, the event is surprising and carries high information.
- We define self-information as $I(y) = -\log_2 p(y)$. The log ensures additivity: information from two independent events is the sum of their individual information ($I(y_1, y_2) = I(y_1) + I(y_2)$).

2. **Entropy**: The entropy $H(Y)$ is simply the **expected value** (average) of the self-information over all possible classes:

$$H(Y) = \mathbb{E}[I(y)] = \sum_{j=1}^{J} p(j) \cdot (-\log_2 p(j)) = -\sum_{j=1}^{J} p(j) \log_2 p(j) \tag{42}$$

Connection to Decision Trees: At a given node, we want to know: *"How many bits on average do I need to specify the class of an example?"*

- If the node is pure (e.g., 100% Class A), $Entropy = -1 \log_2 1 = 0$. We need 0 bits; we already know the answer.
- If the node is a 50/50 mix (Binary), $Entropy = -0.5 \log_2 0.5 - 0.5 \log_2 0.5 = 1$. We need 1 bit to determine the class.

Information Gain represents the **reduction in the number of bits** needed to determine the class after observing the attribute split.

51.4 Handling Attribute Types

- **Nominal (Categorical)**:
  - Multi-way split: One branch per value.
  - Binary split: Group values into two sets (e.g., {Sports, Luxury} vs {Family}). Requires checking $2^{k-1} - 1$ combinations.
- **Ordinal**: Similar to nominal, but grouping must respect the order (e.g., {Low, Medium} vs {High} is valid; {Low, High} vs {Medium} is not).
- **Continuous**:
  - Discretize into bins (Static).
  - Dynamic Binary Split: Test $A < v$. We sort the values and check split points between adjacent values to find the best threshold v.

51.5 Overfitting and Pruning

Decision trees easily overfit. A tree can grow until every single leaf is pure (sometimes containing just one example), effectively memorizing the noise.

- **Pre-Pruning**: Stop growing early (e.g., max depth, min samples per leaf, min impurity decrease). Simple but risks "horizon effect" (stopping before a good split).
- **Post-Pruning**: Grow the full tree, then prune branches bottom-up if removing them (replacing with a leaf) doesn't significantly increase error on validation data.

52 Ensemble Learning

The idea of Ensemble Learning is to train multiple models ("base learners") and combine their predictions. This often results in a model that is more robust and accurate than any single constituent.

52.1 Bagging (Bootstrap Aggregation)

Bagging reduces **variance** and is particularly effective for unstable models (like Decision Trees), which change dramatically with small changes in the data.

52.1.1 Algorithm

1. **Bootstrap Sampling**: Generate M new datasets $D_1, \ldots, D_M$ from the original training set D by sampling N items **uniformly with replacement**. Each D_m has the same size as D but some duplicates (approximately 63% unique).
2. **Train**: Build a model (e.g., a tree) $f_m(\mathbf{x})$ on each D_m.
3. **Aggregate**:
   - **Regression**: Average the outputs: $y = \frac{1}{M} \sum_{m=1}^{M} f_m(\mathbf{x})$.
   - **Classification**: Majority vote.

52.1.2 Random Forest

Random Forest is a specific implementation of Bagging with an extra trick to **decorrelate** the trees.

- Standard Bagging: High probability that strong features are used at the top of every tree, making trees correlated.
- **Feature Bagging**: When building each split in a tree, Random Forest only considers a **random subset** of m features (typically $m \approx \sqrt{M}$ where M is total features).

This forces the trees to learn diverse patterns, further reducing variance when aggregated.

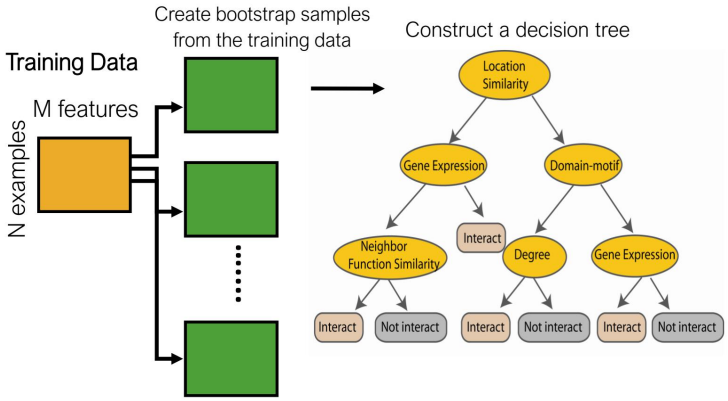


Figure 75: Random Forest conceptually creates many trees and averages their output.

52.2 Boosting

Boosting reduces **bias** and can convert weak learners (slightly better than random) into a strong learner. Unlike Bagging (parallel), Boosting is **sequential**.

52.2.1 Concepts

Models are trained one after another. Each subsequent model focuses on the examples that were **misclassified** by the previous ones.

52.2.2 Conceptual Walkthrough

Imagine you are studying for an exam.

1. **Round 1**: You take a practice test (train Model 1). You understand some topics but fail others.
2. **Round 2**: You study again, but this time you spend effectively **more time** (higher weight) on the questions you got wrong. You train Model 2 specifically to solve these "hard" questions. Model 2 might be bad at the easy stuff, but that's okay.
3. **Round 3**: You look at the mistakes made by *both* Model 1 and Model 2, assign them even higher weights, and train Model 3.

Finally, you take the actual exam (Test Set). You don't just trust the last model; you ask all three. However, you give more "vote" (α_t) to the model that performed best overall during training.

52.2.3 Key Components of AdaBoost

- **Sample Weights (w_i)**: A distribution over the training set. Initially uniform ($1/N$). If a point is misclassified, its weight increases. If correctly classified, its weight decreases.

  How do weights change the training?

  - **Method 1: Weighted Loss Function** (Standard): The weak learner minimizes a weighted error instead of standard error. For a decision tree, this means the Gini Index calculation counts a point with weight w_i as effectively being w_i distinct examples.
  - **Method 2: Resampling**: We create a new dataset by sampling N items from the original data *with probability proportional to* w_i. Hard examples (high w_i) will appear multiple times in the new dataset, forcing the learner to get them right.

- **Classifier Weight (α_t)**: How much we trust the current weak learner.

  - If error $\gamma_t \approx 0$ (excellent), α_t is large very large.
  - If error $\gamma_t \approx 0.5$ (random guessing), $\alpha_t = 0$ (it gets no vote).

52.2.4 AdaBoost Algorithm

AdaBoost is the most popular boosting algorithm. It maintains a set of weights w_i over the training dataset.

Algorithm 3 AdaBoost for Binary Classification $t_i \in \{-1, 1\}$

1: Initialize weights $w_i^{(1)} = 1/N$ for all $i = 1 \dots N$.
2: **for** $t = 1 \dots T$ **do**
3: Train weak learner $f_t(\mathbf{x})$ using weights $\mathbf{w}^{(t)}$.
4: Compute weighted error rate:

$$\gamma_t = \frac{\sum_{i=1}^{N} w_i^{(t)} \mathbb{I}(f_t(\mathbf{x}_i) \neq t_i)}{\sum_{i=1}^{N} w_i^{(t)}}$$

5: Compute classifier coefficient (quality):

$$\alpha_t = \ln\left(\sqrt{\frac{1-\gamma_t}{\gamma_t}}\right) \text{ or } \frac{1}{2}\ln\left(\frac{1-\gamma_t}{\gamma_t}\right)$$

▷ Note: Course slides use $\ln \frac{1-\gamma}{\gamma}$, which is equivalent to $2\times$ standard definition; the scaling doesn't change the sign of the prediction.
6: Update weights:

$$w_i^{(t+1)} \leftarrow w_i^{(t)} \exp\left(\alpha_t \cdot \mathbb{I}(f_t(\mathbf{x}_i) \neq t_i)\right)$$

7: ... or equivalently $w_i^{(t+1)} \leftarrow w_i^{(t)} \exp(-\alpha_t t_i f_t(\mathbf{x}_i))$ if we normalize differently.
8: **end for**
9: **Output**: $Y(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t f_t(\mathbf{x})\right)$

Intuition:

- If $\gamma_t \approx 0$ (good model), α_t is large.
- If $\gamma_t \approx 0.5$ (random guess), $\alpha_t \approx 0$.
- We increase weights for misclassified points so the next learner is forced to "correct" these mistakes.

52.3 Comparison: Bagging vs Boosting

Bagging	**Boosting**
Train models independently (Parallel).	Train models sequentially.
Aims to reduce **Variance** (Overfitting).	Aims to reduce **Bias** (Underfitting).
Robust to outliers.	Sensitive to outliers (tries hard to fit them).
Example: Random Forest	Example: AdaBoost, Gradient Boosting

Table 3: Summary of differences.

53 Lab Session: AdaBoost Implementation

In this lab, we implemented the AdaBoost algorithm from scratch to classify forest cover types using cartographic variables.

53.1 Dataset

We used the **Forest Cover Type** dataset.

- **Goal**: Predict the cover type (categorical).
- **Modification**: We filtered the dataset to include only classes "1" and "2" to create a binary classification problem.
- **Labels**: Mapped to $\{-1, 1\}$ for AdaBoost compatibility.

53.2 Implementation Details

We used 'scikit-learn''s `DecisionTreeClassifier` as the base weak learner. The core training loop involves updating sample weights after each iteration.

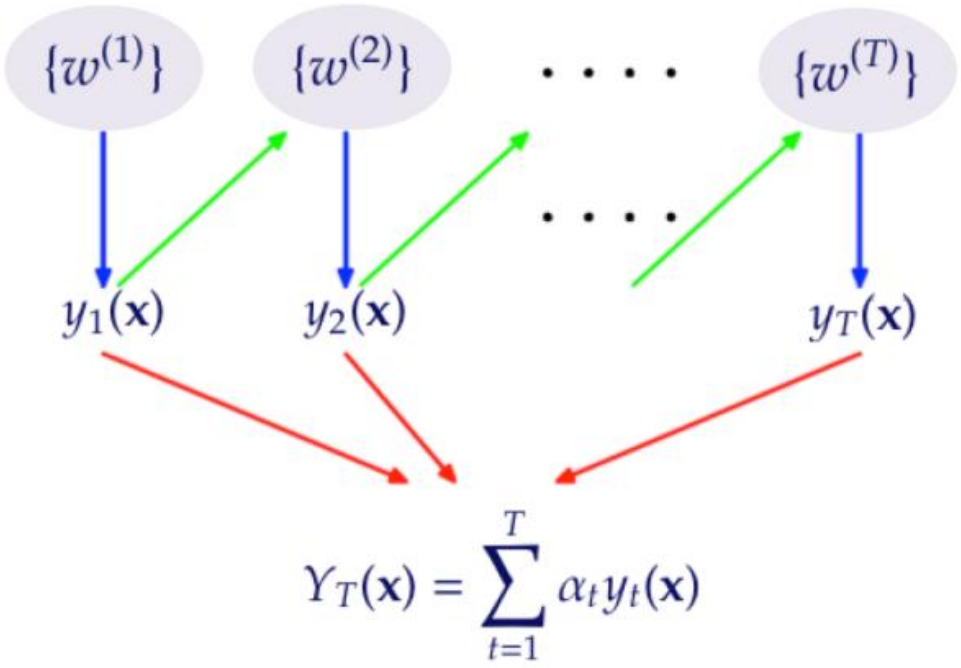


Figure 76: Schematic of the Boosting framework implemented in the lab.

53.2.1 Key Code Snippet

The following Python code demonstrates the weight update step:

```
# Initialize weights
w = np.ones(N) / N

for t in range(T):
```

```
# 1. Train weak learner with current weights
clf = DecisionTreeClassifier(max_depth=D)
clf.fit(X_train, y_train, sample_weight=w)

# 2. Predict and calculate weighted error rate (gamma)
y_pred = clf.predict(X_train)
indicator = np.not_equal(y_pred, y_train) # 1 if error, 0 if correct
gamma = w[indicator].sum() / w.sum()

# 3. Calculate coefficient (alpha)
alpha = 0.5 * np.log((1 - gamma) / gamma) # or simply log((1-g)/g)

# 4. Update weights
# Increase weight for misclassified (indicator=1)
w *= np.exp(alpha * indicator)

# (Optional) Normalize weights
w /= w.sum()
```

53.3 Optimization and Results

We analyzed the effect of the tree depth D on the test error.

- **Shallow Trees ($D = 1$, Decision Stumps)**: High bias individually, but AdaBoost is very effective at combining them into a strong classifier.
- **Deep Trees**: Lower bias but higher variance. If base trees are too complex, they might overfit the weighted data, reducing the benefit of boosting.
- **Observation**: There is an optimal depth (e.g., $D \approx 3 - 5$) that balances the weakness of the learner with the power of the ensemble.

Lecture 9: Unsupervised Learning - Other Topics

Contents

54 Introduction

This lecture covers the fundamentals of Neural Networks, starting with a recap of Ensemble Learning (specifically Boosting and AdaBoost) as a transition into complex learning models. We then explore the Perceptron, the building block of neural networks, its limitations, and how Multi-Layer Perceptrons (MLPs) overcome them. Finally, we derive the Backpropagation algorithm in detail and provide an overview of Deep Learning architectures.

55 Recap: Ensemble Learning

Before diving into neural networks, we revisit Ensemble Learning, contrasting Bagging and Boosting.

55.1 Bagging vs. Boosting

Ensemble methods combine multiple base learners to improve performance. Two primary strategies are:

- **Bagging (Bootstrap Aggregation)**:
  - **Method**: Generates M bootstrap samples (random sampling with replacement) from the training set. A separate model is trained on each sample in parallel.
  - **Aggregation**: For regression, it averages predictions:

$$y_{bag}(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^{M} y_m(\mathbf{x})$$

    For classification, it uses majority voting or averages probabilities.
  - **Goal**: Reduces **variance**. It is effective for unstable classifiers (e.g., fully grown decision trees).
  - **Example**: **Random Forest**.
- **Boosting**:
  - **Method**: Trains classifiers sequentially. Each new classifier focuses on the errors made by previous ones by re-weighting the training data.
  - **Aggregation**: Final prediction is a weighted sum of the component classifiers.
  - **Goal**: Reduces **bias**. It is effective for weak learners (classifiers slightly better than random guessing).
  - **Example**: **AdaBoost** (Adaptive Boosting), Gradient Boosting.

55.2 AdaBoost (Adaptive Boosting)

AdaBoost is a specific boosting algorithm that adjusts weights of training samples.

55.2.1 Algorithm Overview

Given a dataset $D_n = \{(\mathbf{x}_i, t_i)\}_{i=1}^n$ where $t_i \in \{-1, 1\}$.

1. **Initialize weights**: $w_i^{(1)} = 1/n$ for all $i = 1, \ldots, n$.
2. **Iterate** for $t = 1$ to T:
   (a) Train base classifier y_t on D_n weighted by $w^{(t)}$.
   (b) Compute weighted error rate:
   $$\gamma_t = \frac{\sum_{i=1}^n w_i^{(t)} I(y_t(\mathbf{x}_i) \neq t_i)}{\sum_{i=1}^n w_i^{(t)}}$$
   where $I(\cdot)$ is the indicator function (1 if true, 0 otherwise).
   (c) Compute classifier coefficient (quality) α_t:
   $$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\gamma_t}{\gamma_t}\right)$$
   (Note: The slides use $\alpha_t = \ln(\ldots)$, but standard derivation often includes $1/2$. We will stick to the slide's implication or standard form. The slide shows $\alpha_t \leftarrow \ln \ldots$. We will use the standard form to ensure mathematical consistency if needed, but the slide formula $\alpha_t = \ln(\frac{1-\gamma_t}{\gamma_t})$ leads to a specific update rule).
   (d) Update weights:
   $$w_i^{(t+1)} = w_i^{(t)} \exp(\alpha_t I(y_t(\mathbf{x}_i) \neq t_i))$$
   (e) Normalize weights (optional but recommended for numerical stability).
3. **Final Prediction**:
   $$Y_T(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^T \alpha_t y_t(\mathbf{x})\right)$$

55.2.2 Intuition

- If a sample is misclassified ($I = 1$), its weight increases (multiplied by $e^{\alpha_t} > 1$ since $\alpha_t > 0$ for $\gamma_t < 0.5$).
- If a sample is correctly classified ($I = 0$), its weight stays same (or decreases relatively after normalization).
- The next classifier is forced to focus on the "hard" misclassified examples.

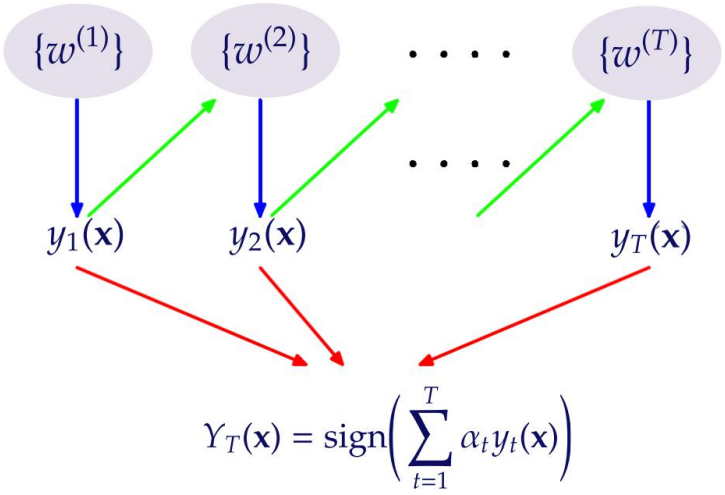


Figure 77: AdaBoost combines weak classifiers (vertical/horizontal lines) to form a complex decision boundary.

56 The Perceptron

The Perceptron (Rosenblatt, 1962) is the simplest type of artificial neural network and a linear binary classifier.

56.1 Model Definition

A perceptron takes an input vector $\mathbf{x} = [1, x_1, \ldots, x_p]^T$ (including bias $x_0 = 1$) and computes a weighted sum. The output is determined by an activation function.

$$f(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}) = \sigma\left(w_0 + \sum_{j=1}^{p} w_j x_j\right)$$

Where:

- $\mathbf{w}$ is the weight vector.
- $\sigma(\cdot)$ is the activation function.

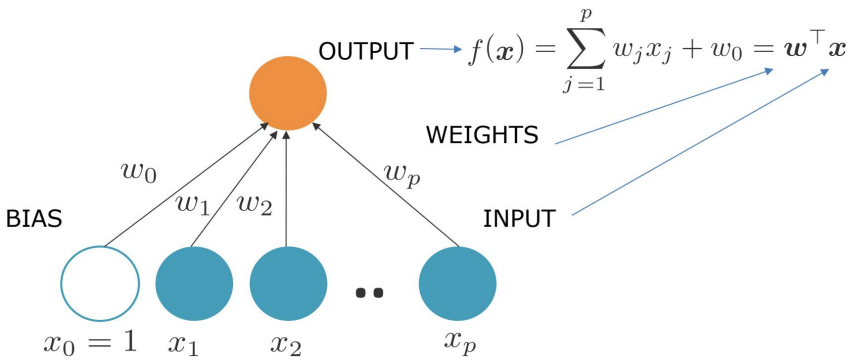


Figure 78: Structure of a Perceptron.

56.2 Activation Functions

The choice of non-linear activation function $\sigma(u)$ is crucial.

1. **Step Function** (Threshold):

$$\sigma(u) = \begin{cases} 1 & \text{if } u > 0 \\ 0 & \text{otherwise} \end{cases}$$

   Used in the original perceptron. Not differentiable at 0.

2. **Sigmoid (Logistic) Function**:

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

   Values in $(0, 1)$. Smooth and differentiable, interpreted as probability.

3. **Hyperbolic Tangent (Tanh)**:

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

   Values in $(-1, 1)$. Zero-centered.

4. **Softmax** (for multi-class):

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

56.3 Loss Functions

To train the network, we treat the problem as an optimization task minimizing a loss function $J(\Theta)$. Common choices include:

1. **Mean Squared Error (MSE)** (for regression):

$$J(\Theta) = \frac{1}{2} \sum_i (y^{(i)} - f(\mathbf{x}^{(i)}))^2$$

2. **Cross-Entropy Loss** (for classification): Used when the output is a probability (e.g., with Sigmoid or Softmax). For binary classification:

$$J(\Theta) = -\sum_i \left[y^{(i)} \log f(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \log(1 - f(\mathbf{x}^{(i)})) \right]$$

   For multi-class classification (as in the Lab), this generalizes to summing over all K classes.

56.4 Training: The Perceptron Learning Algorithm

For the original step-function perceptron, weights are updated iteratively based on errors.

Algorithm:

1. Initialize weights randomly.
2. Select a learning rate η (e.g., $0 < \eta \leq 1$).
3. For each sample $(\mathbf{x}_i, y_i)$:
4. Predict $\hat{y} = \text{step}(\mathbf{w}^T \mathbf{x}_i)$ (where $\text{step}(u) = 1$ if $u > 0$, else 0).
5. If $\hat{y} \neq y_i$:
   - If $y_i = 1$ but $\hat{y} = 0$ (under-prediction): $\mathbf{w} \leftarrow \mathbf{w} + \eta \mathbf{x}_i$
   - If $y_i = 0$ but $\hat{y} = 1$ (over-prediction): $\mathbf{w} \leftarrow \mathbf{w} - \eta \mathbf{x}_i$
6. Repeat until convergence (0 errors on linearly separable data).

56.4.1 Theoretical Derivation (Perceptron Criterion)

While the update rule seems intuitive, it is formally derived as Stochastic Gradient Descent (SGD) on a specific loss function called the **Perceptron Criterion**.

For binary classification with labels $t_i \in \{-1, 1\}$, a point is misclassified if $t_i \mathbf{w}^T \mathbf{x}_i \leq 0$. The loss function is defined as the sum of overlaps for misclassified points $\mathcal{M}$:

$$E_p(\mathbf{w}) = -\sum_{i \in \mathcal{M}} t_i \mathbf{w}^T \mathbf{x}_i$$

Minimizing this error using SGD:

1. Pick a random misclassified example $(\mathbf{x}_i, t_i)$.
2. Compute gradient: $\nabla_{\mathbf{w}} E_p = -t_i \mathbf{x}_i$.
3. Update weight with step size η:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} E_p = \mathbf{w} + \eta t_i \mathbf{x}_i$$

This exactly matches our update rule (if $t_i = 1$, add $\eta \mathbf{x}_i$; if $t_i = -1$, subtract $\eta \mathbf{x}_i$).

56.5 Limitations: The XOR Problem

A single perceptron can only learn **linearly separable** functions. It fails on the XOR problem because no single line can separate the classes $(0,0) \to 0, (1,1) \to 0$ from $(0,1) \to 1, (1,0) \to 1$. This limitation, highlighted by Minsky and Papert (1969), led to the first "AI Winter".

57 Multi-Layer Perceptron (MLP)

To solve non-linear problems like XOR, we stack perceptrons into layers, introducing **hidden layers** with non-linear activation functions.

57.1 Architecture

An MLP consists of:

1. **Input Layer**: Features $\mathbf{x}$.
2. **Hidden Layer**: Consists of H neurons (hidden units). The output z_h of the h-th hidden unit is:
$$z_h = \sigma(\mathbf{w}_h^T \mathbf{x}) \quad \text{for } h = 1, \ldots, H$$
3. **Output Layer**: Final prediction is a weighted sum of the hidden units' outputs:
$$f(\mathbf{x}) = v_0 + \sum_{h=1}^{H} v_h z_h$$

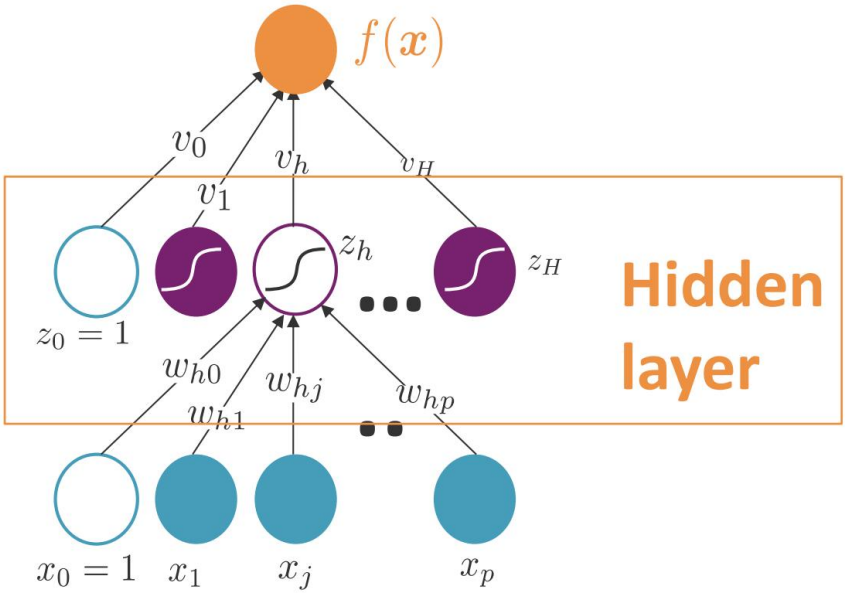


Figure 79: A Multi-Layer Perceptron with one hidden layer.

57.2 Solving XOR with MLP

By introducing a hidden layer, the network effectively transforms the input space into a new space where the classes become linearly separable.

- The hidden units act as feature detectors (e.g., one detects "x1 OR x2", another "x1 NAND x2").
- The output unit combines these to compute XOR.

57.3 Universal Approximation Theorem

A feedforward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function.

58 Backpropagation

Training MLPs is harder than single perceptrons because we don't have "ground truth" targets for the hidden neurons. The **Backpropagation** algorithm (Rumelhart et al., 1986) solves this by propagating the error gradients from the output layer back to the input layer using the **chain rule**.

58.1 Problem Setup

We minimize a loss function $E(\mathbf{w})$ (e.g., Mean Squared Error) using Stochastic Gradient Descent (SGD).

$$E = \frac{1}{2}(y - f(\mathbf{x}))^2$$

Weight update rule:

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

58.2 Derivation of Backpropagation

Consider a network with one hidden layer:

1. **Input**: $\mathbf{x} = [x_1, \ldots, x_p]^T$
2. **Hidden**: $z_h = \sigma(a_h)$ where $a_h = \sum_{j=1}^{p} w_{hj} x_j$ (for $h = 1 \ldots H$)
3. **Output**: $y_{pred} = \sum_{h=1}^{H} v_h z_h$ (linear output)

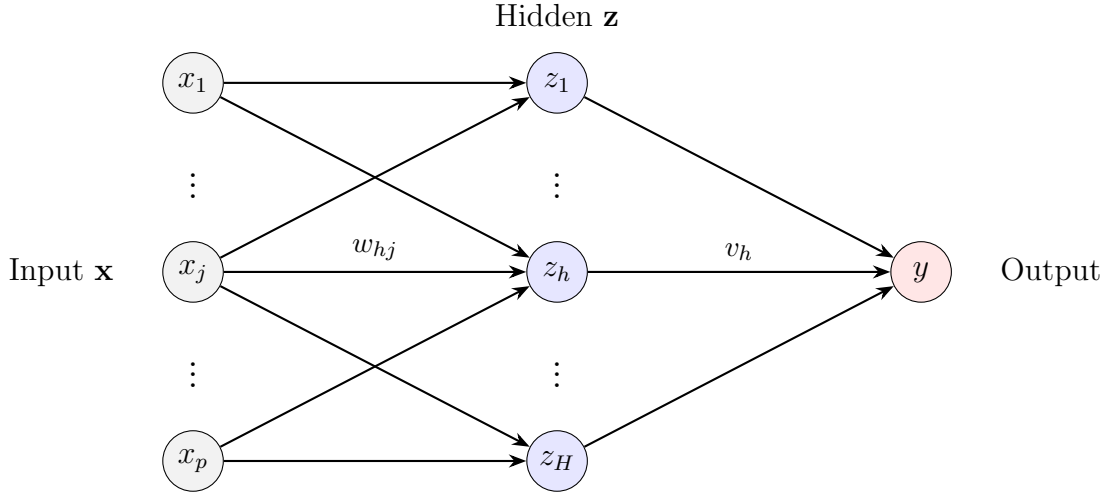


Figure 80: Computation graph for the 1-hidden-layer network. We compute gradients for v_h and w_{hj}.

Let's compute gradients for output weights v_h and hidden weights w_{hj}.

58.2.1 1. Gradient for Output Weights (v_h)

$$\frac{\partial E}{\partial v_h} = \frac{\partial E}{\partial f} \cdot \frac{\partial f}{\partial v_h}$$

Using squared error $E = \frac{1}{2}(y - f)^2$:

$$\frac{\partial E}{\partial f} = -(y - f)$$

$$\frac{\partial f}{\partial v_h} = z_h$$

So:

$$\frac{\partial E}{\partial v_h} = -(y - f)z_h$$

Update: $\Delta v_h = \eta(y - f)z_h$

58.2.2 2. Gradient for Hidden Weights (w_{hj})

We use the chain rule to go deeper:

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial f} \cdot \frac{\partial f}{\partial z_h} \cdot \frac{\partial z_h}{\partial a_h} \cdot \frac{\partial a_h}{\partial w_{hj}}$$

Known terms:

- $\frac{\partial E}{\partial f} = -(y - f)$ (Error at output)
- $\frac{\partial f}{\partial z_h} = v_h$ (Input to output neuron)
- $\frac{\partial z_h}{\partial a_h} = \sigma'(a_h) = z_h(1 - z_h)$ (if σ is sigmoid)
- $\frac{\partial a_h}{\partial w_{hj}} = x_j$ (Input to hidden neuron)

Combining them:

$$\frac{\partial E}{\partial w_{hj}} = -(y - f) \cdot v_h \cdot z_h(1 - z_h) \cdot x_j$$

Define the "error signal" δ:

- Output error signal: $\delta_{out} = (y - f)$
- Hidden error signal: $\delta_h = \delta_{out} v_h \sigma'(a_h)$

Then weight updates are simply:

$$\Delta w_{hj} = \eta \delta_h x_j$$

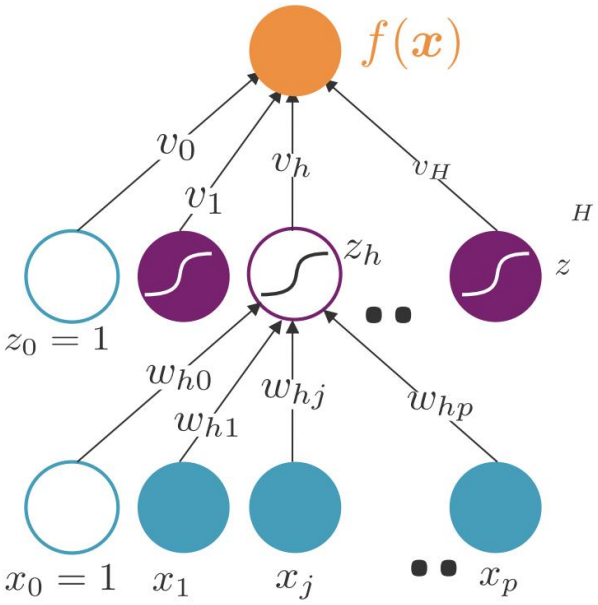


Figure 81: Summary of the Backpropagation Algorithm.

58.3 Implementation Details

- **Initialization**: Weights must be initialized randomly (e.g., small Gaussian noise) to break symmetry. Zero initialization prevents learning in hidden layers.
- **Regularization**: Weight decay (L2 regularization) is added to the loss to prevent overfitting ($E' = E + \frac{\lambda}{2} \sum w^2$).
- **Dropout**: Randomly dropping units during training to improve robustness.

59 Deep Learning Overview

Deep Learning extends MLPs to many layers (Deep Neural Networks) and specialized architectures.

- **Convolutional Neural Networks (CNNs)**: Specialized for grid data like images. Use shared weights and pooling layers to detect spatial features.
- **Recurrent Neural Networks (RNNs) & LSTMs**: Designed for sequential data (text, time series). Maintain hidden state memory.
- **Transformers**: State-of-the-art for NLP and vision, relying on attention mechanisms.
- **Unsupervised**: Autoencoders (data compression), GANs (generative models).

60 Lab Work: Neural Networks for Digit Recognition

In this lab, we implement a Neural Network for handwritten digit recognition using the MNIST dataset.

60.1 The MNIST Dataset

The MNIST database contains 60,000 training and 10,000 testing images of handwritten digits (0-9).

- Each image is 28×28 pixels (grayscale, normalized to [0, 1]).
- Input dimension: 784 features (unrolled pixels).
- Output classes: 10 (digits 0-9).

Figure 82: Example of a handwritten digit (3) from MNIST.

60.2 Neural Network Structure

We use a 3-layer network (Input, Hidden, Output):

$$\mathbf{x} \xrightarrow{\Theta^{(1)}} \mathbf{a}^{(2)} \xrightarrow{\Theta^{(2)}} \mathbf{a}^{(3)}(h_\theta(\mathbf{x}))$$

Where $\Theta^{(l)}$ are the weight matrices.

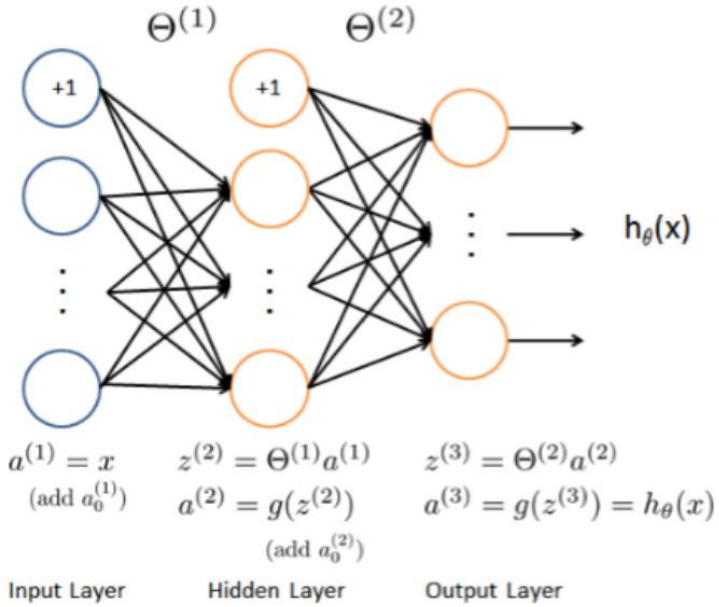


Figure 83: Neural Network architecture with one hidden layer.

60.3 Implementation Tasks

1. **Initialization**: Weights are initialized randomly within $[-\epsilon, \epsilon]$ to break symmetry. Zero initialization would cause all neurons to learn the same features.

2. **Sigmoid Function**: Implement $g(z) = \frac{1}{1+e^{-z}}$ and its gradient $g'(z) = g(z)(1-g(z))$.

3. **Cost Function**: Implement the Cross-Entropy loss (with optional regularization λ):

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[y_k^{(i)} \log(h_\theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\theta(x^{(i)}))_k) \right] + \text{Reg}$$

4. **Backpropagation**: Compute gradients $\nabla_\Theta J(\Theta)$ to update weights using an optimization algorithm (e.g., L-BFGS-B or SGD).

Lecture 10: Introduction to Deep Learning and CNNs

Contents

61 Introduction

This lecture serves as an introduction to Deep Learning, moving from the foundational concepts of Perceptrons and Multi-layer Perceptrons (MLPs) to the more specialized Convolutional Neural Networks (CNNs). We will cover the history and motivation behind Deep Learning, explore the building blocks of neural networks, and dive deep into the architecture and operations of CNNs.

62 Recap: From Perceptrons to Multilayer Perceptrons

Before diving into deep learning, it is essential to revisit the basic building blocks: the Perceptron and the Multilayer Perceptron.

62.1 The Perceptron

The Perceptron, inspired by neurobiology, is a linear discriminant model for binary classification. It computes a weighted sum of inputs and applies a threshold function.

The output y is given by:

$$y = \text{sign}(a_0 x_0 + a_1 x_1 + \cdots + a_p x_p) = \text{sign}(\mathbf{w}^T \mathbf{x}) \tag{43}$$

where $\mathbf{w}$ includes the bias term w_0 (associated with $x_0 = 1$).

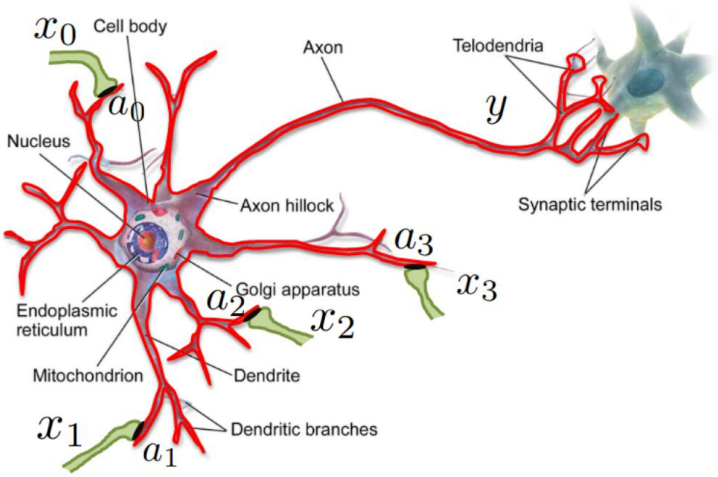


Figure 84: Biological inspiration for the Perceptron.

The decision boundary created by a Perceptron is a hyperplane defined by $\mathbf{w}^T \mathbf{x} = 0$.

62.2 Multiclass Classification

For K classes, we use K outputs. The classification rule is to choose the class k that maximizes the output function $f_k(\mathbf{x})$. To interpret these outputs as probabilities, the **softmax** function is used:

$$P(y = k|\mathbf{x}) = \frac{e^{o_k}}{\sum_{l=1}^{K} e^{o_l}} \quad \text{where} \quad o_k = \mathbf{w}_k^T \mathbf{x} \tag{44}$$

62.3 The Multi-layer Perceptron (MLP)

The MLP overcomes the limitations of the simple Perceptron (which can only solve linearly separable problems) by introducing **hidden layers** with non-linear activation functions.

An MLP with one hidden layer can be described as:

$$f(\mathbf{x}) = v_0 + \sum_{h=1}^{H} v_h z_h \tag{45}$$

where z_h is the output of the hidden unit h:

$$z_h = \sigma(\mathbf{w}_h^T \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}_h^T \mathbf{x}}} \tag{46}$$

Here, σ is the sigmoid activation function.

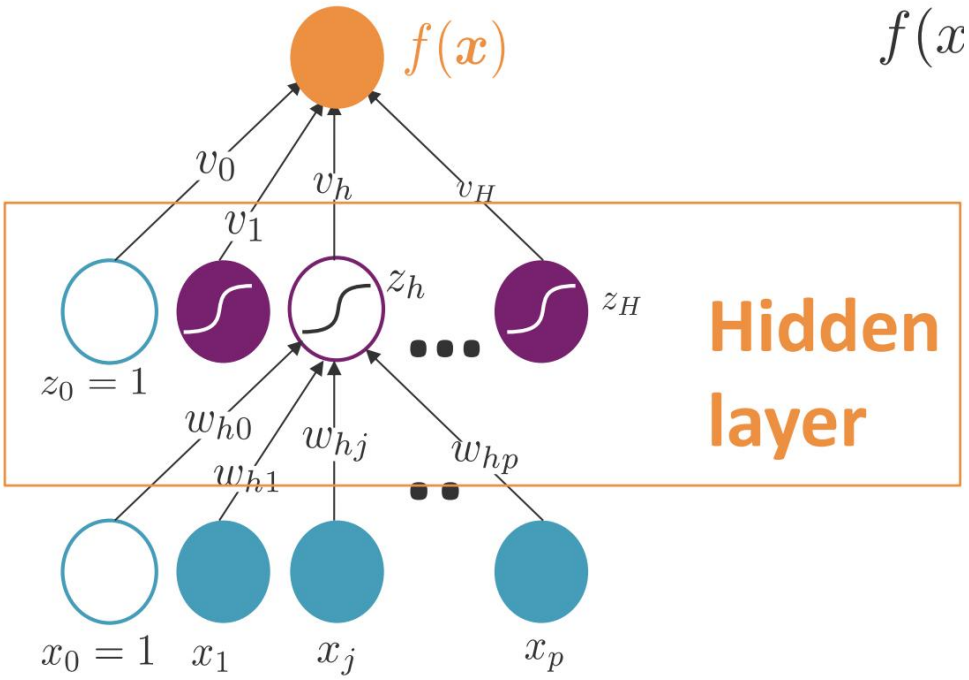


Figure 85: Structure of a Multi-layer Perceptron.

62.4 Backpropagation

Training an MLP involves adjusting the weights to minimize an error function. This is done using **Stochastic Gradient Descent (SGD)**. The key challenge is computing the gradient of the error with respect to weights in early layers. **Backpropagation** solves this by propagating the error backwards from the output layer to the input layer using the chain rule.

The general update rule for a weight w_j is:

$$w_j^{(t+1)} = w_j^{(t)} - \eta \frac{\partial \text{Error}}{\partial w_j} \tag{47}$$

where η is the learning rate.

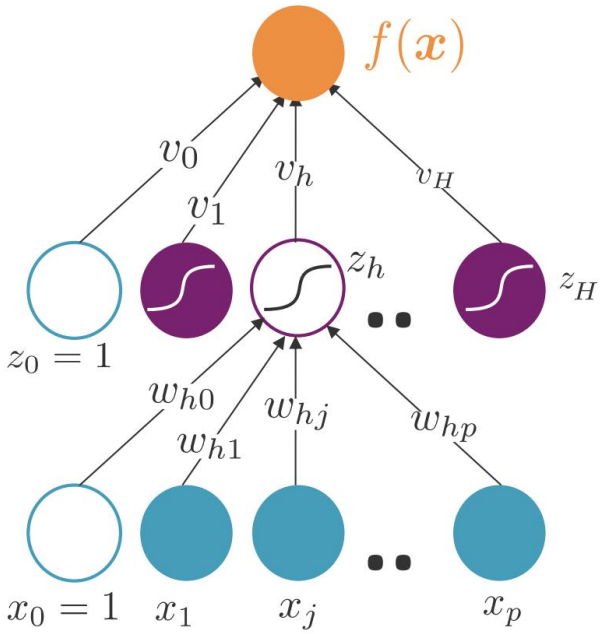


Figure 86: The concept of backward propagation of errors.

The algorithm iterates between:

1. **Forward pass**: Compute activations and outputs.
2. **Backward pass**: Compute errors and gradients.
3. **Update**: Adjust weights.

63 Introduction to Deep Learning

Deep Learning is a subset of Machine Learning that focuses on learning data representations rather than just task-specific algorithms. It uses deep architectures (many layers) to learn hierarchical features.

63.1 Deep Learning Definition

A family of parametric, non-linear, and deep hierarchical representation learning functions optimized with stochastic gradient descent to encode domain knowledge (invariances, stationarity).

Mathematically, a deep network is a composition of functions:

$$a_L(x;\theta) = h_L(h_{L-1}(\dots h_1(x,\theta_1),\theta_{L-1}),\theta_L) \tag{48}$$

Learning involves finding parameters θ^* that minimize a loss function ℓ:

$$\theta^* = \arg\min_{\theta} \sum_{(x,y)\in\mathcal{D}} \ell(y, a_L(x;\theta)) \tag{49}$$

63.2 Why Now?

Deep Learning has seen a resurgence (often called the "Deep Learning Renaissance") due to three main factors:

1. **Faster Computers**: Availability of GPUs (Graphics Processing Units) which are highly efficient for matrix operations.
2. **More Data**: The explosion of digital data (images, text, user interactions) provides the fuel needed to train large models.
3. **Better Software**: Powerful open-source libraries like PyTorch and TensorFlow make implementation accessible.

63.3 Representation Learning

The core power of Deep Learning lies in its ability to learn features automatically.

- **Traditional ML**: Requires manual feature extraction (e.g., SIFT for images, TF-IDF for text) followed by a simple classifier.
- **Deep Learning**: Learns features and the classifier jointly. The initial layers learn simple features (edges, colors), while deeper layers combine them into complex concepts (shapes, objects).

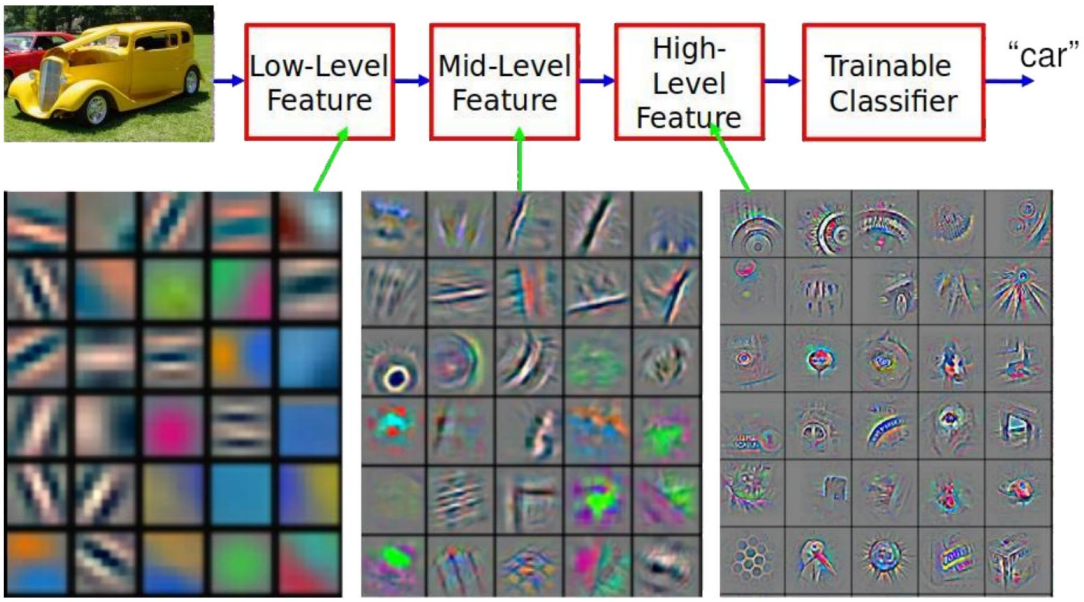


Figure 87: Hierarchical features learned by a CNN: from low-level edges to high-level object parts.

64 Neural Network Modules

A neural network is built from modules. A module is a function $a = h(x, w)$ that takes an input x and parameters w, and returns an output a. Crucially, it must be differentiable to allow for backpropagation.

64.1 Common Modules

64.1.1 Linear Module

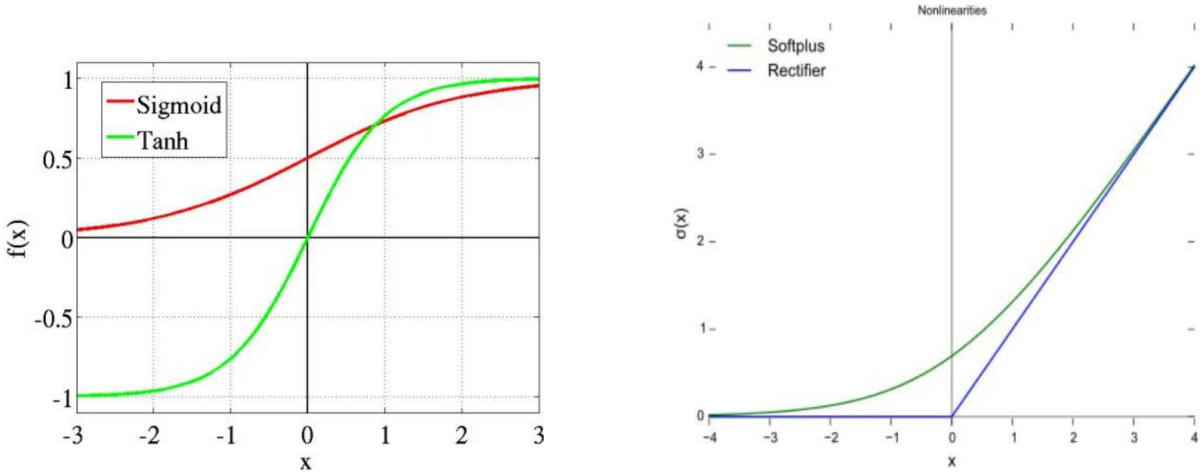
- **Function**: $a = wx$
- **Gradient**: $\frac{\partial a}{\partial w} = x$
- **Pros**: Easy to train, no saturation.
- **Cons**: Cannot capture non-linear relationships.

64.1.2 Sigmoid and Tanh

- **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}} \in [0, 1]$. Used heavily in the 90s, interpretable as probability.
- **Tanh**: $\tanh(x) \in [-1, 1]$. Often performs better than sigmoid due to zero-centered output.
- **Cons**: prone to **vanishing gradients**. When the input is large (positive or negative), the gradient is close to zero, stopping learning in deep networks.

64.1.3 Rectified Linear Unit (ReLU)

- **Function**: $h(x) = \max(0, x)$
- **Gradient**: 1 if $x > 0$, else 0.
- **Pros**:
  - Solves vanishing gradient problem for positive inputs.
  - Computationally efficient (simple thresholding).
  - Sparsity (outputs are exactly 0 for negative inputs).
- **Cons**: Non-differentiable at 0 (handled by convention, e.g., gradient=0). Dead neurons if weights push all inputs to negative.



(a) Sigmoid and Tanh

(b) ReLU

Figure 88: Activation Functions.

65 Convolutional Neural Networks (CNNs)

Standard fully connected networks (MLPs) ignore the spatial structure of images. Treating an image as a flattened vector discards local correlations (e.g., a pixel is closely related to its neighbors). Furthermore, fully connected layers scale poorly with image size (a 200×200 image into 1000 neurons requires 40 million parameters).

CNNs leverage the **spatial structure** of images through:

- **Local Connectivity**: Neurons connect only to a local region of the input.
- **Parameter Sharing**: The same filter (weights) is used across the entire image.
- **Invariance**: Pooling layers provide invariance to small translations.

65.1 The Convolution Operation

Mathematically, the convolution of two functions f and g is defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{50}$$

In the context of discrete images (2D), a convolution with a filter (kernel) K on an image I is:

$$(I * K)[i,j] = \sum_m \sum_n I[i-m, j-n]K[m,n] \tag{51}$$

In deep learning frameworks, this is often implemented as a cross-correlation (no flipping of the kernel), but it's referred to as convolution.

65.1.1 Properties

- **Linearity**: $f * (g + h) = f * g + f * h$
- **Associativity**: $(f * g) * h = f * (g * h)$
- **Commutativity**: $f * g = g * f$

65.1.2 Key Concepts

- **Filter (Kernel)**: A small matrix of weights that slides over the input. It detects specific features like edges, corners, or textures.
- **Feature Map**: The output of convolving a filter with the input.
- **Stride (s)**: The step size the filter moves by. Larger stride reduces output size.
- **Padding (p)**: Adding zeros around the border to preserve spatial dimensions.

65.2 Output Dimensions

Given an input of size $W_{in} \times H_{in} \times D_{in}$ (Volume), a filter of size $F \times F$, stride S, and padding P, the output dimensions are:

$$W_{out} = \frac{W_{in} - F + 2P}{S} + 1 \tag{52}$$

$$H_{out} = \frac{H_{in} - F + 2P}{S} + 1 \tag{53}$$

The depth of the output D_{out} is equal to the number of filters used (K). The depth of the filter itself must match the input depth D_{in}.

65.3 Pooling Layers

Pooling layers reduce the spatial dimensions (downsampling) to reduce computation and control overfitting. They also provide translation invariance.

- **Max Pooling**: Takes the maximum value in a window. Preserves the strongest feature activation.
- **Average Pooling**: Takes the average value (similar to blurring/smoothing an image).

Example of Max Pooling on a 2×2 window:

$$\begin{array}{|c|c|} \hline 1 & 3 \\ \hline 2 & 9 \\ \hline \end{array} \xrightarrow{\text{Max Pool}} \boxed{9}$$

65.4 LeNet Architecture

LeNet (LeCun et al., 1990) was one of the first successful CNNs, designed for digit recognition (MNIST). It consists of alternating convolution and pooling layers, followed by fully connected layers.

The network learns hierarchical features:

- **First Layers**: SImple features like edges and oriented lines.
- **Middle Layers**: Combinations of edges (corners, curves).
- **last Layers**: Complex object parts and whole objects.

65.5 Practical Issues and Invariances

CNNs are designed to be robust to certain variations:

- **Translation Invariance**: Achieved via pooling. An object moving slightly in the input leads to the same pooled output.
- **Scale and Rotation**: Standard CNNs are *not* inherently invariant to large scale or rotation changes. This is often handled by **Data Augmentation** (training the network on rotated/scaled versions of the images).

66 Analysis of Depth

Why go deep?

- **Parameter Efficiency**: Deeper networks can represent complex functions with fewer parameters than wide, shallow networks.
- **Hierarchy**: Depth allows the composition of features (edges $\rightarrow$ shapes $\rightarrow$ objects).

As shown in research (e.g., comparisons on ImageNet), increasing depth (while managing training stability) generally leads to better accuracy, although simply adding layers without techniques like Residual Connections (ResNet) can degrade performance due to optimization difficulties.

67 Lab 9: Convolutional Neural Networks for Digit Recognition

This lab focuses on implementing a Convolutional Neural Network (CNN) using TensorFlow/Keras to solve the handwritten digit recognition task (MNIST dataset). This task has broad applicability, from postal code recognition to bank check processing.

67.1 The MNIST Dataset

The MNIST dataset consists of 60,000 training images and 10,000 testing images of handwritten digits (0-9).

- **Image Size**: 28×28 pixels, grayscale (pixel values 0-255, normalized to [0, 1]).
- **Features**: Flattened, each image has 784 features.
- **Classes**: 10 discrete classes (digits 0-9).

Figure 89: Example of a handwritten digit (3) from the MNIST dataset.

67.2 CNN Pipeline with LeNet

The goal is to implement the LeNet architecture, which typically consists of:

1. **Convolutional Layer**: Extracts spatial features.
2. **Pooling Layer**: Reduces dimensionality and adds invariance.
3. **Convolutional Layer**: Extracts higher-level features.
4. **Pooling Layer**: Further reduction.
5. **Fully Connected Layers**: Classification based on extracted features.

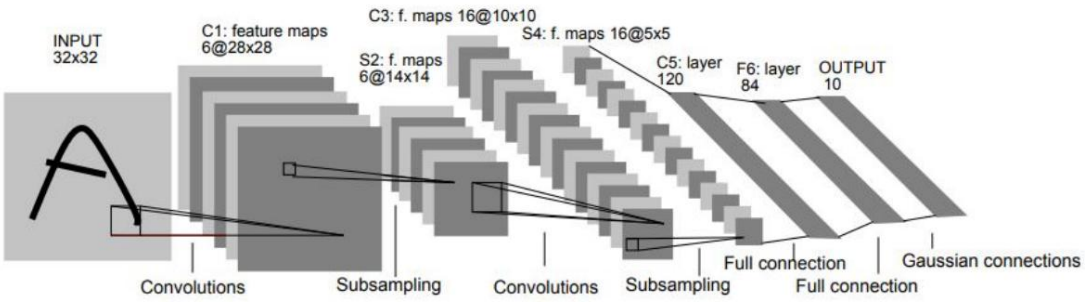


Figure 90: The LeNet Architecture used in the lab.

67.3 Implementation Steps

1. **Environment Setup**: Installing `tensorflow`, `virtualenv`, and `jupyter`.
2. **Data Loading**: Using `tf.keras.datasets.mnist.load_data()` to load train/test sets and `tf.keras.utils.to_categorical` for one-hot encoding labels.
3. **Model Definition**: Using `keras.layers.Conv2D`, `MaxPooling2D`, and `Dense` to build the sequential model.
4. **Cost Function**: Categorical Cross-Entropy is used for multi-class classification:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\theta(x^{(i)})_k) \tag{54}$$

5. **Training**: Using `model.compile` (with optimizer and loss) and `model.fit` (training for a set number of epochs).

67.4 Evaluation

- **Metrics**: Accuracy and Loss on both training and test sets.
- **Confusion Matrix**: Analyzing which digits are most frequently confused (e.g., 4 and 9, or 1 and 7).
- **Visualizing Kernels**: Inspecting the learned filters in the first convolutional layer to understand what features (edges, curves) are being detected.

67.5 Exploration

The lab encourages experimenting with:

- **Activation Functions**: Comparing the standard LeNet's Tanh with the modern ReLU.
- **Regularization**: Adding **Dropout** layers to prevent overfitting.

68 Course Summary

This lecture marks the transition to Deep Learning, the final major topic of the course. We have covered the Machine Learning Pipeline:

1. **Data Pre-processing**: Dimensionality reduction (PCA), Feature selection.
2. **Model Selection**: Bias-variance tradeoff, Cross-validation.
3. **Supervised Learning**: Regression (Linear, Logistic), Classification (kNN, SVM, Decision Trees, Naïve Bayes, Neural Networks).
4. **Unsupervised Learning**: Clustering (k-Means, Spectral).
5. **Ensemble Learning**: Bagging, AdaBoost.