ESLR Notes

Compiled Notes

Contents

Part I: General ML Notes

ewpage

These notes provide a comprehensive foundation in machine learning concepts, combining theoretical understanding with practical intuition. The material is drawn from StatQuest videos and Cornell's CS4780 course, two excellent resources for learning ML.

Purpose and Goals

These notes aim to help you:

1. **Understand the fundamentals**: Build a solid foundation in statistics and probability before diving into ML algorithms
2. **Develop intuition**: Learn not just the "how" but the "why" behind each technique
3. **See connections**: Understand how different algorithms relate to each other
4. **Apply knowledge**: Gain practical insights for implementing these methods

Topics Covered

1. **Basic Statistics**: Sampling, probability, hypothesis testing, confidence intervals—the foundation for all ML
2. **Decision Trees**: Intuitive tree-based models that recursively partition the feature space
3. **Gradient Boosting**: The powerful technique of combining weak learners into a strong ensemble
4. **XGBoost**: The industry-standard implementation of gradient boosting with key optimizations
5. **Clustering**: Unsupervised techniques for discovering structure in unlabeled data
6. **Support Vector Machines**: Maximum margin classifiers with the kernel trick for non-linear boundaries
7. **Dimensionality Reduction**: Techniques like PCA and t-SNE for handling high-dimensional data
8. **Regression**: From simple linear regression to logistic regression for classification

Prerequisites

Before starting, you should be comfortable with:

- Basic algebra and calculus (derivatives, partial derivatives)

- Linear algebra fundamentals (vectors, matrices, dot products)
- Basic probability concepts (events, conditional probability)

How to Use These Notes

- Start with Basic Statistics if you need to refresh foundational concepts
- For each topic, focus on understanding the intuition before the math
- Pay attention to the “why” questions—understanding motivation helps retention
- Work through examples to solidify understanding

Resources

- **StatQuest with Josh Starmer**: Excellent video explanations with clear visualizations
- **Cornell CS4780**: More rigorous treatment of machine learning theory

ewpage

Basic Statistics

Statistics is the foundation of machine learning. Before any model can learn patterns, we need to understand the data itself—how to collect it properly, describe it meaningfully, and make valid inferences from samples to populations.

Sampling and Measurement

When we collect data, we're measuring characteristics (called **variables**) of our subjects:

Types of Variables: - **Quantitative** (Numerical): Things we can measure with numbers (height, temperature, income) - **Categorical** (Qualitative): Things we describe with categories (color, species, disease status)

Measurement Scales—different types require different statistical treatments: - **Interval Scale** (Quantitative): Differences are meaningful, but no true zero (temperature in Celsius—0°C doesn't mean "no temperature") - **Nominal Scale** (Qualitative): Unordered categories (eye color, blood type)—can only check equality - **Ordinal Scale** (Qualitative): Ordered categories (education level, Likert scale ratings)—can rank but can't quantify differences

The Sampling Problem: Any statistic we compute from a sample will vary from sample to sample. This variation comes from two sources: - **Sampling Error**: Unavoidable random variation from using a sample instead of the whole population - **Sampling Bias**: Systematic errors in how we collected the sample - *Selection Bias*: Some population members more likely to be included (surveying only online users) - *Response Bias*: People don't answer truthfully (socially desirable responses) - *Non-Response Bias*: Certain types of people don't respond (busy people skip surveys)

Sampling Methods: - **Simple Random Sampling**: Each data point has equal probability of being selected—the gold standard but sometimes impractical - **Stratified Random Sampling**: Divide population into meaningful subgroups (strata), then sample from each - Ensures representation from all important subgroups - Often produces lower variance estimates than simple random sampling - Example: Stratifying by age groups when studying health outcomes - **Cluster Sampling**: Divide into clusters (often geographic), randomly select clusters, sample within them - More practical for large geographic areas - Example: Randomly selecting cities, then sampling households within those cities - **Multi-Stage Sampling**: Combination of sampling methods at different stages

Descriptive Statistics

Descriptive statistics summarize what the data looks like before we make inferences.

Measures of Central Tendency: - **Mean** ($\bar{x}$): The arithmetic average—sensitive to outliers - **Median**: The middle value when sorted—robust to outliers - **Mode**: The most frequent value—useful for categorical data

Shape of Distribution—understanding where mean and median fall: - **Symmetric**: Mean coincides with median (normal distribution is symmetric) - **Left Skewed** (negatively skewed): Long left tail Mean < Median - Example: Age at retirement (most retire around 65, some retire very young) - **Right Skewed** (positively skewed): Long right tail Mean > Median - Example: Income distribution (most earn moderate amounts, few earn millions) - **Key insight**: The mean is "pulled" toward the long tail

Standard Deviation—measures spread around the mean: - Deviation = how far each observation is from the mean - $s = \sqrt{\frac{\sum(x_i - \bar{x})^2}{N-1}}$ - Why $N-1$? This is "Bessel's correction"—using N would underestimate the true population variance

The 68-95-99.7 Rule (Empirical Rule) for normally distributed data: - ~68% of data falls within 1 standard deviation of the mean - ~95% of data falls within 2 standard deviations - ~99.7% of data falls within 3 standard deviations (the "3-sigma rule")

Interquartile Range (IQR): - IQR = 75th percentile – 25th percentile (the middle 50% of data) - Common outlier rule: Points beyond $1.5 \times$ IQR from Q1 or Q3 are potential outliers

Probability

Probability quantifies uncertainty. Here's how we formalize it:

Expected Value (what you'd "expect" on average): - $E(X) = \sum_i x_i \times p(X = x_i)$ - This is the first moment about the origin—the "center of mass" of the distribution

Variance (how spread out the values are): - $V(X) = E(X^2) - (E(X))^2 = E[(X-\mu)^2]$ - This is the second moment about the mean

Z-score (standardization): - $z = \frac{y-\mu}{\sigma}$ - Transforms any normal distribution to the **Standard Normal Distribution** $\sim N(0,1)$ - Interpretation: "How many standard deviations away from the mean?"

Covariance and Correlation: - Covariance: $Cov(X,Y) = E[(X-\mu_x)(Y-\mu_y)]$—direction of linear relationship - Correlation: $\rho = \frac{Cov(X,Y)}{\sigma_x \sigma_y} = E(z_x z_y)$—standardized covariance, ranges from -1 to +1

The Central Limit Theorem (CLT)—one of the most important results in statistics: - Sample means follow an approximately normal distribution regardless of the population distribution - $\bar{X} \sim N(\mu, \frac{\sigma}{\sqrt{N}})$ - **Standard Error** = $\frac{\sigma}{\sqrt{N}}$—the standard deviation of the sampling distribution - Key insight: Standard error decreases with $\sqrt{N}$, not N (need 4x samples to halve error)

Example: Exit Poll Survey - Binary outcome (vote A or B), assume $p = 0.5$, sample size $N = 1800$ - Standard deviation: $\sqrt{p(1-p)} = 0.5$ - Standard error: $\frac{0.5}{\sqrt{1800}} \approx 0.012$ - 99% CI: $0.5 \pm 3 \times 0.012 \approx (0.47, 0.53)$

Example: Income Survey - Population: $\sim N(380, 80^2)$ (mean $380K, SD $80K), sample size

$N = 100$ - Question: What's $P(\bar{y} \geq 400)$? - Standard error: $\frac{80}{\sqrt{100}} = 8$ - $z = \frac{400-380}{8} = 2.5$ - $P(Z \geq 2.5) < 0.006$ (very unlikely to see sample mean ≥ 400 by chance)

Confidence Interval

Confidence intervals quantify our uncertainty about parameter estimates.

Point Estimate: A single number as our best guess (e.g., sample mean $\bar{x}$ for population mean μ)

Properties of Good Estimators: - **Unbiased**: $E(\bar{X}) = \mu$ (on average, the estimator equals the true value) - **Efficient/Consistent**: $N \to \infty \implies V(\bar{X}) \to 0$ (variance shrinks as sample grows)

Interval Estimate: A range of plausible values - CI = Point Estimate ± Margin of Error - **Confidence Level**: The probability (e.g., 95%) that the procedure produces an interval containing the true parameter - Important: A 95% CI doesn't mean "95% probability the parameter is in this interval"—the parameter is fixed, not random!

CI for Proportions: - Point estimate: $\hat{\pi}$ - Standard error: $\sqrt{\frac{\hat{\pi}(1-\hat{\pi})}{N}}$ - 95% CI: $\hat{\pi} \pm z_{0.025} \times se$ where $z_{0.025} \approx 1.96$

CI for Means: - Point estimate: $\bar{X}$ - When population variance is unknown, use sample variance and **t-distribution** instead of z-distribution - The t-distribution has heavier tails—accounts for extra uncertainty from estimating variance - As $N \to \infty$, t-distribution approaches normal distribution - CI: $\bar{X} \pm t_{n-1,\alpha/2} \times \frac{s}{\sqrt{N}}$

Sample Size Determination: - For proportions: $N = \frac{\pi(1-\pi)\times z^2}{M^2}$ where M is desired margin of error - For means: $N = \frac{\sigma^2 \times z^2}{M^2}$ - Note: Quadrupling sample size only halves the margin of error!

Estimation Methods: - **Maximum Likelihood Estimation (MLE)**: Find parameter values that maximize the probability of observing the data we actually observed - **Bootstrap**: Resample from observed data to estimate standard errors and CIs—no distributional assumptions needed

Significance Test

Hypothesis testing provides a framework for making decisions based on data.

Five Components of a Significance Test:

1. **Assumptions**: What conditions must hold?
    - Type of data, randomization, population distribution, sample size
2. **Hypotheses**:
    - H_0 (Null): The "no effect" or "status quo" hypothesis
    - H_1 (Alternative): What we're testing for
3. **Test Statistic**: Quantifies how far the observed data falls from what H_0 predicts

4. **P-value**: Probability of observing data as extreme or more extreme than what we got, *assuming* H_0 *is true*
    - Small p-value □ data is unlikely under H_0 □ evidence against H_0
    - P-value is NOT the probability that H_0 is true!
5. **Conclusion**: "Reject H_0" or "Fail to reject H_0"
    - Note: We never "accept H_0"—absence of evidence is not evidence of absence

Testing Proportions: - $H_0 : \pi = \pi_0$ vs $H_1 : \pi \neq \pi_0$ - Test statistic: $z = \frac{\hat{\pi} - \pi_0}{se}$ where $se = \sqrt{\frac{\pi_0(1-\pi_0)}{N}}$

Testing Means: - $H_0 : \mu = \mu_0$ vs $H_1 : \mu \neq \mu_0$ - Test statistic: $t = \frac{\bar{X} - \mu_0}{s/\sqrt{N}}$ - For small samples, use exact binomial distribution

One-Tailed vs Two-Tailed Tests: - One-tailed: Tests deviation in one direction only - More powerful for detecting effects in that direction - Risky: Can't detect effects in the opposite direction - Use only with strong prior justification

Types of Errors: - **Type I Error** (False Positive): Reject H_0 when it's actually true - Probability = significance level (α, often 0.05) - **Type II Error** (False Negative): Fail to reject H_0 when it's actually false - Probability denoted β - **Power** = $1 - \beta$ = probability of correctly rejecting false H_0

Trade-offs: - Decreasing Type I error increases Type II error - The closer the true parameter is to H_0, the lower the power - Larger samples □ more power

Important Warning: Statistical significance ≠ practical significance. With large enough samples, tiny, meaningless differences become "statistically significant."

Comparison of Groups

Most research involves comparing groups—treatments vs control, different populations, etc.

Difference in Means (Independent Samples): - Goal: Test if $\mu_1 - \mu_2 = 0$ - Estimate: $\bar{y}_1 - \bar{y}_2$ - Standard error: $se = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$ - CI: $(\bar{y}_1 - \bar{y}_2) \pm t \times se$ - Test statistic: $t = \frac{(\bar{y}_1 - \bar{y}_2) - 0}{se}$ with $df = n_1 + n_2 - 2$

Equal Variance Assumption (Pooled Variance): - If we assume $\sigma_1 = \sigma_2$, we can pool the data to get a better variance estimate: - $s_{pooled} = \sqrt{\frac{(n_1-1)s_1^2 + (n_2-1)s_2^2}{n_1+n_2-2}}$ - $se = s_{pooled}\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$

Difference in Proportions: - Standard error: $se = \sqrt{\frac{\hat{\pi}_1(1-\hat{\pi}_1)}{n_1} + \frac{\hat{\pi}_2(1-\hat{\pi}_2)}{n_2}}$ - For significance test, pool proportions under H_0: - $\hat{\pi}_{pooled}$ and $se = \sqrt{\hat{\pi}(1-\hat{\pi})\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}$

Paired/Matched Samples: - Same subject measured at different times (before/after) - Controls for between-subject variation - Analyze the *differences* directly - Test: $t = \frac{\bar{d} - 0}{s_d/\sqrt{n}}$ - Example: Blood pressure before and after treatment for the same patients

Effect Size: Standardized measure of the magnitude of difference - Cohen's d: $d = \frac{\bar{y}_1 - \bar{y}_2}{s_{pooled}}$ - Interpretation: Small (~0.2), Medium (~0.5), Large (~0.8)

McNemar Test (for paired proportions):

	Treatment Yes	Treatment No
Control Yes	n_{11}	n_{12}
Control No	n_{21}	n_{22}

- Test statistic: $z = \frac{n_{12} - n_{21}}{\sqrt{n_{12} + n_{21}}}$
- Only the discordant pairs (n_{12} and n_{21}) matter!

Non-Parametric Alternatives (when normality is violated): - **Wilcoxon Signed-Rank Test**: For paired data—ranks the absolute differences - **Mann-Whitney U Test**: For independent samples—compares ranks between groups

Association between Categorical Variables

When both variables are categorical, we analyze their relationship through contingency tables.

Statistical Independence: Variables are independent if knowing one tells you nothing about the other. Mathematically: $P(A|B) = P(A)$

Chi-Square Test: 1. Calculate expected frequencies under independence: $f_e = \frac{\text{row total} \times \text{column total}}{\text{grand total}}$ 2. Compare to observed frequencies: $\chi^2 = \sum \frac{(f_o - f_e)^2}{f_e}$ 3. Degrees of freedom: $(r-1) \times (c-1)$

Important: χ^2 tells you *if* there's an association, not *how strong* it is!

Residual Analysis—which cells drive the association? - Standardized residual: $z = \frac{f_o - f_e}{\sqrt{f_e(1-\text{row\%})(1-\text{col\%})}}$ - $|z| > 2$ suggests the cell is significantly different from expected

Odds Ratio (2×2 tables): - $\theta = \frac{n_{11} \times n_{22}}{n_{12} \times n_{21}}$ - Interpretation: - $\theta = 1$: No association (equal odds) - $\theta > 1$: Higher odds for row 1 - $\theta < 1$: Lower odds for row 1 - Example: If odds ratio for smoking and lung cancer is 10, smokers have 10× the odds of lung cancer

Ordinal Variables—when categories have a natural order: - **Concordant pairs**: Higher on X corresponds to higher on Y - **Discordant pairs**: Higher on X corresponds to lower on Y - **Gamma coefficient**: $\gamma = \frac{C - D}{C + D}$ - Ranges from -1 to +1, like correlation for ordinal data

P-value Interpretation and Common Misconceptions

Understanding what the p-value actually means is crucial:

What p-value IS: - The probability of obtaining a test statistic at least as extreme as observed, *given that the null hypothesis is true* - A measure of how compatible the data is with the null hypothesis

What p-value is NOT: - □ The probability that the null hypothesis is true - □ The probability that results occurred “by chance” - □ The probability of making a Type I error (that’s α, which you set beforehand)

Guidelines for Interpretation: - Small p-value (e.g., < 0.05): Data is unlikely under H_0—evidence against H_0 - Large p-value: Data is compatible with H_0—but doesn’t prove H_0 is true - Always report effect sizes alongside p-values - Consider practical significance, not just statistical significance

ewpage

Decision Trees

Decision trees are among the most intuitive machine learning algorithms. They make predictions by asking a series of yes/no questions about the features, essentially building a flowchart for decision-making. This mirrors how humans often make decisions—through a sequence of simple questions.

Decision Trees

The Core Idea: Recursively split the input/feature space using simple rules (called "stubs"). Each split divides the data into more homogeneous groups.

Key Characteristics: - Splits are always parallel to the feature axes (like drawing vertical or horizontal lines) - Each path from root to leaf represents a conjunction of conditions - The tree structure naturally captures feature interactions

Mathematical Representation: - Each leaf defines a region: $R_j = \{x : d_1 \leq t_1, d_2 \geq t_2, ...\}$ - Prediction for a point: $\hat{Y}_i = \sum_j w_j I\{x_i \in R_j\}$ - Leaf weights (for regression): $w_j = \frac{\sum_i y_i I\{x_i \in R_j\}}{\sum_i I\{x_i \in R_j\}}$ (just the average of y values in that leaf)

Types of Decision Trees:

Binary Splits (most common): - **CART** (Classification and Regression Trees): The most widely used, always binary splits - **C4.5**: Successor to ID3, handles continuous attributes, missing values

Multi-way Splits: - **CHAID** (Chi-Square Automatic Interaction Detection): Uses statistical tests for splits - **ID3**: Original algorithm, only handles categorical features

Splitting

The key question: How do we decide which feature to split on and where?

For Classification Trees—Measuring Impurity:

We want each split to create more "pure" nodes (nodes dominated by one class).

Gini Impurity: - Gini $= 1 - \sum_C p_i^2$ - Intuition: The probability of misclassifying a randomly chosen element if we labeled it randomly according to the class distribution in the node - For a given class i: Probability of picking class i and misclassifying it = $p_i \times (1 - p_i)$ - Sum across all

classes: $\sum_C p_i(1-p_i) = 1 - \sum_C p_i^2$ - Range: 0 (pure node, all one class) to $(K-1)/K$ for K classes (max 0.5 for binary) - Example: If a node has 50% class A and 50% class B, Gini = $1 - (0.5^2 + 0.5^2) = 0.5$

Entropy Criterion: - Based on information theory—measures uncertainty - If event E is very likely ($P(E) \approx 1$): No surprise when it happens - If event E is unlikely ($P(E) \approx 0$): Huge surprise when it happens - Information content: $I(E) = \log(1/P(E)) = -\log(P(E))$ - Entropy = expected information content: $H(E) = -\sum P(E) \log P(E)$ - Range: 0 (pure) to $\log_2(K)$ for K classes (max 1 for binary) - Maximum entropy when all outcomes equally likely

For Regression Trees—Measuring Error: - **Sum of Squared Errors (SSE)**: $\sum_i (Y_i - \bar{Y})^2$ - This is just the variance times N within the node - We want splits that minimize the total SSE across child nodes

Finding the Best Split:

For each candidate split: 1. Calculate the weighted average reduction in impurity/error 2. Weights = number of observations flowing to each child node

Example of Gini Reduction: - Starting Gini at root: $\text{Gini}_{\text{Root}}$ with N_{Root} samples - After split into Left and Right: - $\text{Gini}_{\text{New}} = \frac{N_{\text{Left}}}{N_{\text{Root}}} \times \text{Gini}_{\text{Left}} + \frac{N_{\text{Right}}}{N_{\text{Root}}} \times \text{Gini}_{\text{Right}}$ - Choose the split that minimizes Gini_{New} - Note: $\text{Gini}_{\text{New}} \leq \text{Gini}_{\text{Root}}$ always (splits never increase impurity)

The Algorithm: Greedy search through all features and all possible split points to find the best split at each node.

Bias-Variance Trade-off

Understanding this trade-off is essential for all of machine learning.

Bias: - Measures how well the algorithm can model the true relationship - High bias = making strong/restrictive assumptions - Example: Using a linear model for a parabolic relationship - Low bias = fewer assumptions, more flexible

Variance: - Measures how much the model changes across different training datasets - High variance = model is very sensitive to the specific training data - Low variance = model is stable across different samples

Irreducible Error (Bayes Error): - The inherent noise in the data - Cannot be reduced no matter how good the model

The Trade-off: - Expected Error $=$ Bias^2 $+$ Variance $+$ Irreducible Error - Simple models: High bias, low variance (underfit) - Complex models: Low bias, high variance (overfit) - Goal: Find the sweet spot that minimizes total error

Decision Trees and the Trade-off: - Deep trees have **low bias** (can fit complex patterns) - But **high variance** (very sensitive to training data) - This makes them prone to **overfitting**, especially with: - Noisy samples - Small data samples in deep nodes

Tree Pruning addresses overfitting by adding a complexity penalty: - Objective: Tree Score $=$ SSR $+ \alpha T$ - Where T = number of leaves, α = complexity parameter - As the tree grows, SSR reduction must offset the complexity cost

Nature of Decision Trees

Strengths:

Non-linear Relationships: - Decision trees naturally model complex, non-linear decision boundaries - Unlike splines, which add indicator variables but require continuous boundaries - Trees can create completely discontinuous predictions

No Feature Scaling Required: - Tree algorithms only care about the ordering of values, not their magnitude - No need to normalize or standardize features

Robustness to Outliers: - For input feature outliers: Splits simply ignore extreme values - For output outliers (in regression): Some impact, but less than linear regression - Note: High-leverage points in regression can have extreme influence; trees are more robust

Weaknesses:

Extrapolation: - Trees cannot extrapolate beyond the range of training data - For values outside training range, prediction = nearest leaf's value - Linear models can extrapolate (for better or worse)

Time Series: - Trees cannot capture linear trends or seasonality naturally - Each leaf is a constant prediction—no notion of "trend"

Bagging

Bootstrap Aggregation (Bagging) is a technique to reduce variance.

The Bootstrap: - Given a dataset of size N - Create a new dataset by sampling N points *with replacement* - Probability that point i is never selected: $(1 - \frac{1}{N})^N \approx \frac{1}{e} \approx 0.37$ - So each bootstrap sample contains ~63% of unique original points

Bagging for Trees: 1. Create many bootstrap samples 2. Fit a tree to each 3. Average predictions (regression) or vote (classification)

Why It Works: - Individual trees are unstable (high variance) - Averaging independent predictions reduces variance - If predictions were perfectly independent: Variance would decrease by factor of n

Random Forest

Random Forest = Bagging + Random Feature Selection

The Algorithm: 1. Create bootstrap samples (the "random" part of the data) 2. At each split, consider only a random subset of features: - Classification: typically $\sqrt{p}$ features - Regression: typically $p/3$ features 3. Combine predictions: majority vote (classification) or average (regression)

Why Random Feature Selection?: - Without it, all trees would be very similar (correlated) - If one strong feature dominates, all trees split on it first - Random selection decorrelates the trees

Variance Reduction Math: - Let $\hat{y}_i$ be prediction from tree i, with variance σ^2 - Let ρ be the correlation between trees - Variance of average: $V\left(\frac{1}{n}\sum_i \hat{y}_i\right) = \rho\sigma^2 + \frac{1-\rho}{n}\sigma^2$ - As $n \to \infty$:

Variance approaches $\rho\sigma^2$ - Lower correlation ρ □ lower variance □ better!

Bias vs Variance: - Bias remains the same as a single tree (no improvement) - Variance decreases with more trees - This is why random forests are so powerful: low bias AND low variance

Out-of-Bag (OOB) Error: - ~37% of data not used in each tree (OOB samples) - Use these to estimate test error—like free cross-validation! - For each point, average predictions from trees that didn't train on it - OOB error typically close to leave-one-out cross-validation error

Proximity Matrix: - For OOB observations, count how often each pair lands in the same leaf - Creates a similarity measure between observations - Useful for clustering, visualization, missing value imputation

ExtraTrees

Extremely Randomized Trees—taking randomization even further.

Key Differences from Random Forest:

Aspect	Random Forest	ExtraTrees
Data sampling	Bootstrap (63%)	Entire dataset (100%)
Split thresholds	Optimized search	Randomly selected
Feature selection	Random subset	Random subset

Algorithm: 1. Use the full training set (no bootstrapping) 2. At each node, for each candidate feature: - Select a random threshold uniformly between min and max - Evaluate the split 3. Choose the best feature-threshold combination

Trade-off: - Even more randomness □ even lower variance - But slightly higher bias than Random Forest - Much faster training (no optimization of thresholds)

Variable Importance

Understanding which features matter is often as important as making predictions.

Split-Based Importance (built into tree algorithms): - For each feature j, sum the Gini/entropy reduction across all splits using j - Alternative: Count the number of times feature is used for splitting - **Limitation**: Biased toward continuous features (more possible split points) - **Limitation**: Biased toward high-cardinality categorical features

Permutation-Based Importance (model-agnostic): 1. Calculate baseline accuracy on OOB samples 2. For each feature j: - Randomly shuffle feature j's values (breaks its relationship with target) - Calculate new accuracy - Importance = decrease in accuracy 3. Average across all trees

Why Permutation Importance is Better: - Measures actual predictive value, not just usage - Accounts for redundancy: If feature j has a good surrogate, permuting j won't hurt much - Like setting the coefficient to 0 in regression

Partial Dependence Plots (PDPs): - Show the marginal effect of a feature on predictions - Algorithm: For each value x_s of feature s: - Set all observations to have x_s for that feature - Average the

predictions: $\widehat{f}(x_s) = \frac{1}{N}\sum_i f(x_s, x_{i,-s})$ - **Assumption**: Features are not correlated (can be misleading otherwise) - Can identify interactions using Friedman's H-statistic

Other Importance Methods: - **SHAP (Shapley Values)**: Game-theoretic approach, model-agnostic, handles interactions - **LIME**: Local interpretable model-agnostic explanations—explains individual predictions

Handling Categorical Variables

Binary Categorical: Easy—just a yes/no split

Multi-Category Variables—Options:

One-Hot Encoding: - Create a binary feature for each category - Pro: Simple, works with any algorithm - Con: Increases dimensionality; for trees, biases toward these features

Label Encoding: - Assign ordinal numbers (1, 2, 3, …) - Pro: No dimensionality increase - Con: Imposes an artificial ordering

Native Handling (in tree algorithms): - Consider all possible subsets of categories for binary splits - CART: Finds optimal binary grouping - C4.5, CHAID: Can create multi-way splits (one branch per category) - Pro: Optimal splits; Con: Exponential search space

Tree Pruning

Pruning prevents overfitting by limiting tree complexity.

Pre-Pruning (Early Stopping): - Stop growing before the tree is fully expanded - Criteria: - Maximum depth - Minimum samples per leaf - Minimum impurity decrease - Maximum leaf nodes - Pro: Fast, simple - Con: Might stop too early (a bad split might enable good later splits)

Post-Pruning (Grow then Prune): - Grow a full tree, then remove unhelpful branches - Methods: - **Cost-Complexity Pruning** (CART): Minimize SSE $+ \alpha \times (\text{number of leaves})$ - **Reduced Error Pruning (REP)**: Remove nodes that don't improve validation error - **Pessimistic Error Pruning (PEP)**: Use statistical adjustments on training error - Pro: Considers the full tree structure - Con: More computationally expensive

Selecting the Pruning Level: - Use cross-validation to find optimal α (complexity parameter) - Plot training/validation error vs. α to visualize the trade-off - The "right" amount of pruning balances underfitting and overfitting

ewpage

Boosting

Boosting is one of the most powerful ideas in machine learning: take many “weak” models that are only slightly better than random guessing, and combine them into a “strong” model that achieves high accuracy. It’s like combining many rough rules of thumb into a sophisticated decision-making system.

Overview

The Key Insight: Ensemble methods combine multiple models for better predictions.

Two Main Ensemble Paradigms: - **Bagging**: Build models in parallel on different data subsets, then average - Reduces variance (covered in Decision Trees notes) - **Boosting**: Build models sequentially, each one focusing on previous mistakes - Reduces bias

Boosting Formulation: - $F(x_i) = \sum_m \alpha_m f_m(x_i)$ - f_m = the m-th weak learner (typically a small tree) - α_m = weight given to that learner - Each f_m and α_m are fit jointly, considering what came before

PAC Learning Framework: - PAC = Probably Approximately Correct - Question: Is a problem “learnable”? - A model is PAC-learnable if we can achieve error < ϵ with probability > $(1 - \delta)$ - **Strong learner**: Achieves low error with high probability (but complex, needs lots of data) - **Weak learner**: Only slightly better than random guessing

Schapire’s Breakthrough (1990): “The Strength of Weak Learnability” - Key theorem: If a problem can be solved by a strong learner, it can be solved by combining weak learners - Original mechanism (three hypotheses): - H1: Train on complete data - H2: Train on a balanced sample of H1’s correct and incorrect predictions - H3: Train on examples where H1 and H2 disagree - Final: Majority vote of H1, H2, H3 - Improved performance but couldn’t scale easily □ led to AdaBoost

AdaBoost (Adaptive Boosting): - Construct many hypotheses (not just three) - Key innovation: Sample weights “adapt” based on performance - Correctly classified: weight decreases (pay less attention) - Incorrectly classified: weight increases (focus on mistakes)

Weight Update Mechanism: - Learner weight: $\alpha_m = \frac{1}{2} \log \left[\frac{1-\epsilon_m}{\epsilon_m} \right]$ - Where ϵ_m = weighted classification error - Sample weights: - Correctly classified: $w_i \leftarrow w_i \times e^{-\alpha}$ - Incorrectly classified: $w_i \leftarrow w_i \times e^{\alpha}$

Common Pitfalls: - **Underfitting**: Not enough weak learners in the ensemble - **Overfitting**: Using learners that are too complex (not “weak” enough)

Gradient Boosting

Gradient Boosting generalizes boosting to any differentiable loss function.

The Key Idea: Instead of reweighting samples, fit each new learner to the *negative gradient* of the loss function. The gradient tells us how to "fix" the current predictions.

Why Gradients?: - Gradients point in the direction of steepest increase in loss - Negative gradient = direction to decrease loss most rapidly - The gradient for each point is a proxy for "how poorly is this point being predicted?"

Connection to Gradient Descent: - Regular gradient descent: Update *parameters* in the negative gradient direction - Gradient boosting: Add a *new function* that approximates the negative gradient - Think of it as gradient descent in "function space"

Mathematical Derivation:

We want to minimize loss by adding a new function f_m: - Current: $F(x_i) = \sum_{k=1}^{m-1} \alpha_k f_k(x_i)$ - Goal: Find f_m to minimize $L(F(x_i) + \alpha f_m(x_i))$

Taylor Approximation (first-order): - $L(F + \alpha f_m) \approx L(F) + \alpha f_m \cdot \frac{\partial L}{\partial F}$ - The first term is constant; we minimize the second term - We want: min $\sum_i \frac{\partial L}{\partial F(x_i)} \times \alpha f(x_i)$

Pseudo-Residuals: - Define: $r_i = -\frac{\partial L}{\partial F(x_i)}$ (the negative gradient) - Goal becomes: min $-\sum_i r_i \times \alpha f(x_i)$ - The ensemble improves as long as $\sum_i r_i f(x_i) > 0$

Why "Pseudo-Residuals"?: - For squared loss: $L = \frac{1}{2}(y - F)^2$ - Gradient: $\frac{\partial L}{\partial F} = -(y - F)$ - So $r_i = y_i - F(x_i)$ = actual residual! - For other losses, r_i is *like* a residual (hence "pseudo")

Adapting for CART: - Decision trees minimize squared error naturally - Transform the objective: - min $\sum r_i^2 - 2\sum_i r_i \times \alpha f(x_i) + \sum(\alpha f(x_i))^2$ - min $\sum(r_i - \alpha f(x_i))^2$ - Now the tree can simply minimize squared error between predictions and pseudo-residuals!

Optimal Step Size (Line Search): - $\alpha^* = \frac{\sum r_i f(x_i)}{\sum f(x_i)^2} \approx 1$

The Gradient Boosting Algorithm:

1. **Initialize** with a constant value: $F_0(x) = \arg\min_\gamma \sum L(y_i, \gamma)$
   - For squared loss: just the mean of y
2. **For** $m = 1$ to M:
   a. Compute pseudo-residuals: $r_{im} = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$
   b. Fit a tree to (x_i, r_{im})
   c. For each leaf j, compute optimal output: $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_j} L(y_i, F_m(x_i) + \gamma)$
   d. Update: $F_{m+1}(x) = F_m(x) + \nu \sum_j \gamma_{jm} I(x \in R_j)$

The Shrinkage Parameter (ν): - Also called learning rate - Prevents overfitting by taking small steps - Required because Taylor approximation only works for small changes - Typical values: 0.01 to 0.3

Extension to Classification

For classification, we predict log-odds and minimize negative log-likelihood.

Log-Odds to Probability: - $p = \frac{e^{\log(\text{odds})}}{1+e^{\log(\text{odds})}} = \frac{1}{1+e^{-\log(\text{odds})}}$

The Loss Function (Negative Log-Likelihood): - NLL $= -\sum[y_i \log(p_i) + (1-y_i)\log(1-p_i)]$ - Rewriting in terms of log-odds: - NLL $= -\sum[y_i \cdot \text{log-odds} - \log(1+e^{\text{log-odds}})]$

Computing Pseudo-Residuals: - $\frac{\partial \text{NLL}}{\partial \log(\text{odds})} = p_i - y_i$ - So $r_{im} = y_i - p_i$ (intuitive: how far off is our probability estimate?)

Finding Optimal Leaf Values: - For log-loss, minimizing within each leaf isn't straightforward - Use second-order Taylor approximation: - $\gamma^* = \frac{\sum(y_i - p_i)}{\sum p_i(1-p_i)}$ - Numerator: sum of residuals (first derivative) - Denominator: sum of $p(1-p)$ (second derivative, the Hessian)

The Classification Algorithm:

1. **Initialize**: Log-odds that minimizes NLL (e.g., $\log(\frac{\bar{y}}{1-\bar{y}})$)
2. **For** $m = 1$ to M:
   a. Compute residuals: $r_{im} = y_i - p_i$
   b. Fit a tree to (x_i, r_{im})
   c. For each leaf j: $\gamma_{jm} = \frac{\sum_{i \in R_j}(y_i - p_i)}{\sum_{i \in R_j} p_i(1-p_i)}$
   d. Update: $F_{m+1}(x) = F_m(x) + \nu \sum_j \gamma_{jm} I(x \in R_j)$
3. **Predict**: Convert final log-odds to probability

Gradient Boosting vs AdaBoost

Aspect	AdaBoost	Gradient Boosting
Focus	Reweighting misclassified samples	Fitting negative gradient of loss
Loss function	Exponential loss	Any differentiable loss
Sample weights	Explicitly updated	Implicitly through gradients
Optimization	Coordinate descent	Gradient descent in function space

Key insight: AdaBoost is a special case of Gradient Boosting with exponential loss!

Common Loss Functions

For Regression: - **L2 (Squared Error)**: $L(y, F) = \frac{1}{2}(y-F)^2$ - Gradient = $y - F$ (the actual residual) - Sensitive to outliers - **L1 (Absolute Error)**: $L(y, F) = |y-F|$ - Gradient = $\text{sign}(y-F)$

- More robust to outliers - **Huber Loss**: Combines L1 and L2 - Behaves like L2 for small errors, L1 for large errors - Best of both worlds

For Classification: - **Log Loss**: $L(y, F) = -y \log(p) - (1 - y) \log(1 - p)$ - Standard for probability estimation - **Exponential Loss**: $L(y, F) = e^{-yF}$ - What AdaBoost minimizes - Very sensitive to outliers

Regularization in Gradient Boosting

Learning Rate/Shrinkage (ν): - Scales contribution of each tree - Smaller ν □ need more trees, but better generalization - Trade-off: Training time vs. accuracy

Subsampling (Stochastic Gradient Boosting): - Use only a fraction of data for each tree (e.g., 50-80%) - Similar to mini-batch gradient descent - Reduces overfitting and training time

Early Stopping: - Monitor validation performance - Stop adding trees when validation error stops improving - Prevents overfitting without explicit regularization

Tree Constraints: - Maximum depth (typically 3-8 for boosting) - Minimum samples per leaf - Maximum leaf nodes - Shallow trees = weak learners = less overfitting

AdaBoost for Classification

Let's derive AdaBoost from scratch to understand its mechanics.

Setup: - Binary classification: $y \in \{-1, +1\}$ - Exponential loss: $L(y_i, f(x_i)) = e^{-y_i f(x_i)}$ - This is an upper bound on 0-1 loss

Why Exponential Loss?: - If correct prediction: $y_i f(x_i) > 0$ □ small loss - If wrong prediction: $y_i f(x_i) < 0$ □ exponentially large loss - Forces the algorithm to focus hard on mistakes

Objective Function: - Ensemble: $F(x) = \sum_m \alpha_m f_m(x)$ - Loss: $L = \sum_i e^{-y_i F(x_i)}$

At Round m: - $L = \sum_i e^{-y_i \sum_{k=1}^{m} \alpha_k f_k(x)}$ - $L = \sum_i e^{-y_i \sum_{k=1}^{m-1} \alpha_k f_k(x)} \cdot e^{-y_i \alpha_m f_m(x)}$ - Let $w_i^m = e^{-y_i F_{m-1}(x_i)}$ (weights from previous rounds) - $L = \sum_i w_i^m \cdot e^{-y_i \alpha_m f_m(x_i)}$

Finding Optimal α_m: - Split into correct and incorrect predictions: - $L = \sum_{\text{correct}} w_i e^{-\alpha_m} + \sum_{\text{incorrect}} w_i e^{\alpha_m}$ - Let ϵ_m = weighted misclassification rate - $L = (1 - \epsilon_m) e^{-\alpha_m} + \epsilon_m e^{\alpha_m}$ - Taking derivative and setting to zero: - $\alpha_m^* = \frac{1}{2} \log \left[\frac{1-\epsilon_m}{\epsilon_m} \right]$

Interpreting α_m: - If $\epsilon_m = 0$ (perfect): $\alpha_m \to \infty$ (trust this learner completely) - If $\epsilon_m = 0.5$ (random): $\alpha_m = 0$ (ignore this learner) - If $\epsilon_m > 0.5$ (worse than random): $\alpha_m < 0$ (flip predictions)

The AdaBoost Algorithm:

1. **Initialize**: $w_i = 1/N$ for all samples
2. **For** $m = 1$ to M:
    a. Fit weak learner f_m minimizing weighted error

   b. Compute weighted error: $\epsilon_m = \frac{\sum_i w_i I(y_i \neq f_m(x_i))}{\sum_i w_i}$
   c. Compute learner weight: $\alpha_m = \frac{1}{2} \log \left[\frac{1-\epsilon_m}{\epsilon_m}\right]$
   d. Update sample weights: $w_i \leftarrow w_i \cdot e^{\alpha_m I(y_i \neq f_m(x_i))}$
   e. Normalize weights to sum to 1
3. **Final prediction**: sign $\left(\sum_m \alpha_m f_m(x)\right)$

LogitBoost: Similar to AdaBoost but minimizes logistic loss - $\log(1 + e^{-y_i f(x_i)})$ - More robust to noise and outliers than exponential loss

Notes

Why Boosting Works: - Weak learners have high bias, low variance - Boosting gradually reduces bias by focusing on mistakes - Each iteration adds a small amount of complexity

Historical Development: 1. AdaBoost (1995) - Freund & Schapire 2. AdaBoost interpreted as gradient descent (1999) - Friedman et al. 3. Generalized to any gradient descent (Gradient Boosting)

Gradient Descent vs Gradient Boosting:

Gradient Descent	Gradient Boosting
Updates model *parameters*	Updates model *predictions*
Parameters: $\theta \leftarrow \theta - \eta \nabla L$	Predictions: $F \leftarrow F + \nu f_m$
Fixed model structure	Adds new functions

Gradient Boosting is a Meta-Model: - It's not a single model but a *framework* for combining weak learners - The weak learner can be any model (trees are most common) - The loss function can be customized for different tasks

ewpage

XGBoost

XGBoost (eXtreme Gradient Boosting) is arguably the most successful machine learning algorithm for structured/tabular data. It's a highly optimized implementation of gradient boosting that has won countless Kaggle competitions. What makes it special? Regularization to prevent overfitting, computational tricks for speed, and smart handling of missing values.

Mathematical Details

The Objective Function:

XGBoost adds explicit regularization to the gradient boosting objective:

$$\text{Objective} = \sum_i L(y_i, \hat{y}_i) + \underbrace{\gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2}_{\text{Regularization}}$$

Where: - $L(y_i, \hat{y}_i)$ = Loss function (e.g., MSE, log-loss) - T = Number of leaves in the tree - w_j = Output value (weight) of leaf j - γ = Penalty per leaf (controls tree complexity) - λ = L2 regularization on leaf weights

Common Loss Functions: - **MSE** (Regression): $L = \frac{1}{2}(y_i - \hat{y}_i)^2$ - **Log-loss** (Classification): $L = -[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$

Prediction Update:

$$\hat{y}_i = \hat{y}_i^{(0)} + \sum_{j=1}^{T} w_j \cdot I(x_i \in R_j)$$

Where $\hat{y}_i^{(0)}$ is the prediction from previous rounds.

Second-Order Taylor Approximation:

The key insight: Approximate the loss with a second-order Taylor expansion for efficient optimization.

For adding a new tree with output value O:

$$L(y_i, \hat{y}_i^{(0)} + O) \approx L(y_i, \hat{y}_i^{(0)}) + g_i \cdot O + \frac{1}{2} H_i \cdot O^2$$

Where: - $g_i = \frac{\partial L}{\partial \hat{y}_i}$ (gradient, first derivative) - $H_i = \frac{\partial^2 L}{\partial \hat{y}_i^2}$ (Hessian, second derivative)

Why Second-Order?: - First-order (like regular gradient boosting): Only tells direction - Second-order: Also tells curvature, enabling more accurate steps - Newton's method vs gradient descent—converges faster!

Simplified Objective:

After substitution and dropping constants:

$$\text{Objective} = \sum_i g_i \cdot O + \gamma T + \frac{1}{2}\left(\sum_i H_i + \lambda\right) O^2$$

Optimal Leaf Value:

Taking derivative with respect to O and setting to zero:

$$O^* = -\frac{\sum g_i}{\sum H_i + \lambda}$$

For Different Loss Functions:

Loss	g_i	H_i
MSE	$\hat{y}_i - y_i$	1
Log-loss	$\hat{y}_i - y_i$	$\hat{y}_i(1-\hat{y}_i)$

The Role of λ: - Appears in denominator: $O^* = -\frac{\sum g_i}{\sum H_i + \lambda}$ - High λ □ pushes leaf values toward 0 - Prevents any single leaf from having extreme predictions - Acts like L2 regularization in ridge regression

Regression

Similarity Score (for evaluating potential splits):

$$\text{Similarity} = \frac{G^2}{H + \lambda} = \frac{(\sum g_i)^2}{\sum H_i + \lambda}$$

For MSE loss:

$$\text{Similarity} = \frac{(\sum r_i)^2}{N + \lambda}$$

Where $r_i = y_i - \hat{y}_i$ is the residual and N is the number of samples.

Intuition: - Numerator: Total residual squared (how much signal?) - Denominator: Count + regularization (how much noise?) - Higher similarity = more "pure" node (good for prediction)

Effect of λ: - Large λ □ smaller similarity scores - Reduces sensitivity to individual observations - More pruning, simpler trees

Gain from a Split:

$$\text{Gain} = \text{Similarity}_{\text{left}} + \text{Similarity}_{\text{right}} - \text{Similarity}_{\text{parent}}$$

Split Criterion: - Only split if Gain $> \gamma$ - γ controls minimum improvement required - Even with $\gamma = 0$, pruning still happens (due to regularization)!

Pruning Mechanisms: - Maximum depth - Minimum cover (sum of Hessians, i.e., N for regression) - Trees are grown fully, then pruned backward - Key difference from pre-pruning: A "bad" split might enable good subsequent splits

Final Predictions: - Leaf output: $\frac{\sum r_i}{N+\lambda}$ - Ensemble: Initial Prediction + $\eta \times$ (Tree 1 output) $+ \eta \times$ (Tree 2 output) $+ \ldots$ - Initial prediction = mean of target values - η = learning rate (typically 0.01-0.3)

Classification

For classification, XGBoost works with log-odds (logits) and uses log-loss.

Similarity Score:

$$\text{Similarity} = \frac{(\sum r_i)^2}{\sum p_i(1-p_i) + \lambda}$$

Where: - $r_i = y_i - p_i$ (residual: actual minus predicted probability) - p_i = current probability estimate - Denominator: $\sum p_i(1-p_i)$ comes from the Hessian of log-loss

Gain Calculation: Same as regression.

Cover/Minimum Weight: - For regression: Just N (number of samples) - For classification: $\sum p_i(1-p_i)$ - This is the sum of Hessians—measures "effective sample size" - Points with $p \approx 0.5$ contribute most (most uncertain)

Leaf Output:

$$O = \frac{\sum r_i}{\sum p_i(1-p_i) + \lambda}$$

Ensemble Prediction: 1. Initial prediction = $\log\left(\frac{\bar{y}}{1-\bar{y}}\right)$ (log-odds of class proportion) 2. Add tree contributions with learning rate 3. Final output is log-odds 4. Convert to probability: $p = \frac{1}{1+e^{-\text{log-odds}}}$

Optimizations

XGBoost's speed comes from several clever tricks:

Approximate Greedy Algorithm: - Exact algorithm: Try every possible split point (slow for large data) - Approximate: Bucket continuous features into quantiles - Only consider bucket boundaries as split candidates - Much faster with minimal accuracy loss

Quantile Sketch Algorithm: - Need to find quantiles in a distributed setting - XGBoost uses weighted quantiles (weighted by Hessian/cover) - Points with higher uncertainty (higher H_i) get more weight - More granular splits where they matter most

Sparsity-Aware Split Finding: - Real data often has missing values - XGBoost learns optimal direction for missing values: 1. Compute split gain sending missing values left 2. Compute split gain sending missing values right 3. Choose direction with higher gain - This "default direction" is learned during training - Also works for zero values in sparse data

Cache-Aware Access: - Gradients and Hessians stored in cache for fast access - Block structure for efficient memory access - Out-of-core computation for data that doesn't fit in memory

Comparisons

XGBoost: - Stochastic gradient boosting (row/column subsampling) - No native handling of categorical variables (need encoding) - Depth-wise tree growth (all nodes at same depth split together) - Level-by-level: Explores all possibilities at each depth

LightGBM: - **GOSS** (Gradient-based One-Side Sampling): Oversample high-gradient points - Native encoding for categorical variables - **EFB** (Exclusive Feature Bundling): Combines mutually exclusive features - Histogram-based splitting (faster) - **Leaf-wise growth**: Splits the leaf with highest gain - Faster convergence but can overfit more easily

CatBoost: - **MVS** (Minimum Variance Sampling): More statistically sound sampling - Superior categorical encoding (ordered target encoding to prevent leakage) - **Symmetric trees**: All nodes at same depth use the same split - Faster inference, natural regularization - Handles missing values and categorical features natively

XGBoost vs. Traditional Gradient Boosting

System Optimizations: - **Parallelization**: Tree construction parallelized (not tree-to-tree, but within trees) - **Cache-aware access**: Block structure for efficient memory usage - **Out-of-core computation**: Can handle datasets larger than memory

Algorithmic Enhancements: - **Regularization**: Built-in L1 and L2 on weights - **Missing values**: Learned default directions - **Newton boosting**: Second-order optimization (faster convergence) - **Weighted quantile sketch**: Approximate split finding

Result: XGBoost is often 10-100x faster than sklearn's GradientBoostingClassifier while achieving similar or better accuracy.

Handling Missing Values

The Problem: Most ML algorithms require complete data, forcing imputation.

XGBoost's Approach: - Treat missing values as a special category - During training: Learn whether missing □ left or missing □ right - The "default direction" is chosen to maximize gain - No preprocessing needed!

Why This Works Better: - Imputation (mean, median) assumes missing is similar to observed - XGBoost learns what missing values *actually* mean - Different features can have different optimal directions

Comparison to Traditional Approaches: | Method | Approach | Limitation | |—–|—–|—–| | Mean imputation | Replace with mean | Reduces variance | | Indicator variable | Add “is_missing” feature | Doubles features | | XGBoost native | Learn optimal direction | None—learned from data |

Hyperparameter Tuning

Key Hyperparameters:

Parameter	Description	Typical Range
`n_estimators`	Number of trees	100-10000
`learning_rate` (η)	Step size shrinkage	0.01-0.3
`max_depth`	Maximum tree depth	3-10
`min_child_weight`	Minimum sum of Hessians in leaf	1-10
`gamma` (γ)	Minimum gain for split	0-5
`subsample`	Fraction of rows per tree	0.5-1.0
`colsample_bytree`	Fraction of features per tree	0.5-1.0
`lambda` (λ)	L2 regularization	0-10
`alpha` (α)	L1 regularization	0-10

Tuning Strategy:

1. **Fix learning rate low** (e.g., 0.1), tune other params
2. **Control complexity**: `max_depth`, `min_child_weight`, `gamma`
3. **Add randomness**: `subsample`, `colsample_bytree`
4. **Add regularization**: `lambda`, `alpha`
5. **Lower learning rate** and increase `n_estimators`

Common Approaches: - **Grid Search**: Exhaustive but expensive - **Random Search**: Often just as good, much faster - **Bayesian Optimization**: Intelligent exploration of parameter space - **Early Stopping**: Use validation set, stop when performance plateaus

Rule of Thumb: - Lower `learning_rate` + more `n_estimators` = better but slower - Start with defaults, then tune `max_depth` and `learning_rate` - Use cross-validation to avoid overfitting to validation set

ewpage

Clustering

Clustering is the task of grouping similar objects together without being told what the groups should be. It's **unsupervised learning**—we have no labels, only data. The algorithm discovers structure on its own.

Why Clustering Matters: - Customer segmentation: Group customers by behavior - Image segmentation: Group pixels by color/texture - Document organization: Group articles by topic - Anomaly detection: Normal patterns vs. outliers - Data compression: Represent data by cluster centers

Hierarchical Agglomerative Clustering

The Idea: Start with each point as its own cluster, then repeatedly merge the two most similar clusters until only one remains.

Algorithm: 1. Start: Each data point is its own cluster 2. Compute all pairwise distances between clusters 3. Merge the two closest clusters 4. Repeat steps 2-3 until one cluster remains

This produces a dendrogram—a tree showing the merge history. Cut the tree at any height to get that many clusters.

Defining "Similarity" Between Clusters:

Linkage Method	Definition	Properties
Single Link	Distance between two *closest* members	Tends to create long, chain-like clusters
Complete Link	Distance between two *farthest* members	Creates compact, spherical clusters
Average Link	Average distance between all pairs	Balance between single and complete
Ward's Method	Increase in total within-cluster variance	Minimizes variance, popular choice

When to Use: - When you don't know the number of clusters ahead of time - When you want to see hierarchical structure - For small to medium datasets (doesn't scale well)

Limitation: Very slow! $O(n^3)$ time complexity makes it impractical for large datasets.

K-Means Clustering

The Idea: Partition data into exactly K clusters, each represented by its centroid (center of mass).

Algorithm: 1. Choose K initial cluster centers (randomly or using K-Means++) 2. **Assign** each point to nearest center: $z_n^* = \arg\min_k ||x_n - \mu_k||^2$ 3. **Update** centers as mean of assigned points: $\mu_k = \frac{1}{N_k} \sum_{n:z_n=k} x_n$ 4. Repeat steps 2-3 until assignments don't change

Objective Function (Distortion/Inertia):

$$L = \sum_n ||x_n - \mu_{z_n}||^2$$

This is the total squared distance from each point to its assigned center.

Key Properties: - Each iteration decreases (or maintains) the objective - Guaranteed to converge, but not necessarily to global optimum - Converges quickly in practice

The Initialization Problem: - K-Means is sensitive to initial centers - Bad initialization □ stuck in poor local minimum - Solution: **Multiple restarts**—run many times, keep best result

K-Means++ Initialization (the smart way): 1. Choose first center randomly from data points 2. For each subsequent center: - Compute distance $D(x)$ from each point to nearest existing center - Choose new center with probability proportional to $D(x)^2$ 3. This spreads out initial centers—points far from existing centers more likely to be chosen

Result: Much better starting point, often finds better solutions.

K-Medoids Algorithm (PAM - Partitioning Around Medoids): - Centers must be actual data points (medoids), not means - More robust to outliers - Assignment: $z_n^* = \arg\min_k d(x_n, \mu_k)$ (any distance metric) - Update: Find point with smallest total distance to all other cluster members - **Swap step**: Try swapping current medoid with non-medoid, keep if cost decreases

Choosing the Number of Clusters (K):

This is one of the hardest problems in clustering!

Elbow Method: 1. Plot distortion vs. K 2. Look for "elbow" where distortion stops decreasing rapidly 3. Often subjective—the elbow isn't always clear

Silhouette Score: For each point i: - a_i = average distance to other points in same cluster (cohesion) - b_i = average distance to points in nearest other cluster (separation) - $S_i = \frac{b_i - a_i}{\max(a_i, b_i)}$

Interpretation: - $S_i \approx 1$: Point is well-clustered - $S_i \approx 0$: Point is on boundary between clusters - $S_i \approx -1$: Point may be in wrong cluster

Average silhouette score across all points measures overall clustering quality.

K-Means is EM with Hard Assignments: - Expectation-Maximization (EM) uses "soft" assignments (probabilities) - K-Means uses "hard" assignments (0 or 1) - Both assume spherical clusters with equal variance - K-Means is a special case of Gaussian Mixture Models

Limitations of K-Means: - **Assumes spherical clusters**: Can't find elongated or irregular shapes - **Assumes equal-sized clusters**: Tends to split large clusters - **Sensitive to outliers**: Means are

pulled by extreme values - **Requires specifying** K: Must know number of clusters beforehand - **Non-convex clusters**: Fails on clusters with complex shapes - **Euclidean distance**: May not be appropriate for all data types

Interpreting Results: - **Cluster centers**: "Prototypical" members, useful for understanding what each cluster represents - **Cluster sizes**: Imbalanced sizes might indicate poor clustering or natural structure - **Within-cluster variance**: Measures homogeneity - **Between-cluster variance**: Measures separation

Spectral Clustering

The Idea: Use the eigenvalues (spectrum) of a similarity graph to find clusters. Works when K-Means fails on non-convex shapes.

Graph Perspective: - Data points are nodes - Edges connect similar points (weighted by similarity) - Goal: Find a partition that minimizes edges cut between groups

Algorithm: 1. Build similarity graph from data (e.g., k-nearest neighbors or Gaussian similarity) 2. Compute the Graph Laplacian: $L = D - A$ - D = degree matrix (diagonal, D_{ii} = sum of edges from node i) - A = adjacency matrix (edge weights) 3. Find eigenvectors of L corresponding to smallest eigenvalues 4. Use these eigenvectors as new features, run K-Means

Why the Laplacian?: - Smallest eigenvalue is always 0 (corresponding to all-ones vector) - Second smallest eigenvalue (Fiedler value) indicates best cut - For K clusters, use the K smallest eigenvectors

When to Use: - Non-convex cluster shapes (crescents, rings) - When you have a natural similarity/distance matrix - Graph-structured data

DBSCAN

Density-Based Spatial Clustering of Applications with Noise

The Idea: Clusters are dense regions separated by sparse regions. Points in sparse regions are "noise."

Key Concepts: - **Core Point**: Has at least `minPts` points within radius ϵ - **Border Point**: Within ϵ of a core point, but not itself core - **Noise Point**: Neither core nor border

Reachability: - **Direct Density Reachable**: Point q is within ϵ of core point p - **Density Reachable**: There's a chain of core points connecting p to q - **Density Connected**: Both p and q are density reachable from some core point

Algorithm: 1. For each point, determine if it's a core point 2. Create clusters by connecting core points that are within ϵ of each other 3. Assign border points to nearby clusters 4. Mark remaining points as noise

Parameters: - ϵ (epsilon): Neighborhood radius - `minPts`: Minimum points to form dense region

Choosing Parameters: - `minPts`: Often set to dimensionality $+ 1$ or higher - ϵ: Plot k-distance graph (distance to k-th nearest neighbor), look for elbow

Advantages: - No need to specify number of clusters - Finds arbitrarily shaped clusters - Robust to outliers (identifies them as noise) - Only two intuitive parameters

Disadvantages: - Struggles with clusters of varying density - Parameter selection can be tricky - Doesn't work well in high dimensions (curse of dimensionality) - Can't handle clusters that are close together

Extensions: - **OPTICS** (Ordering Points To Identify Clustering Structure): - Creates an ordering of points based on density - Can extract clusters at different density levels - More robust to parameter choices

- **HDBSCAN** (Hierarchical DBSCAN):
  - Automatically finds clusters of varying densities
  - Combines benefits of DBSCAN and hierarchical clustering
  - More robust, fewer parameters to tune
  - Often the best choice in practice

Choosing a Clustering Algorithm

Decision Guide:

Situation	Recommended Algorithm
Know number of clusters, spherical shapes	K-Means
Don't know number, want hierarchy	Hierarchical Agglomerative
Arbitrary shapes, want to identify noise	DBSCAN or HDBSCAN
Non-convex shapes, know number	Spectral Clustering
Large dataset	K-Means or Mini-batch K-Means
Want to visualize cluster relationships	Hierarchical with dendrogram

Validation Approaches: - **Internal metrics** (no ground truth): Silhouette score, Davies-Bouldin index, Calinski-Harabasz index - **External metrics** (with ground truth): Adjusted Rand Index, Normalized Mutual Information - **Stability**: Do clusters persist with data perturbations?

Remember: Clustering is exploratory—there's often no "right" answer. Different algorithms reveal different structure. The best choice depends on your data and goals.

ewpage

Support Vector Machines

Support Vector Machines (SVMs) are powerful classifiers based on a beautifully geometric idea: find the hyperplane that separates classes with the **maximum margin**. This margin acts as a "safety buffer" that often leads to excellent generalization.

Linear SVM

The Core Question: Given labeled data, what's the "best" hyperplane to separate the classes?

The SVM Answer: The best hyperplane is the one that maximizes the distance to the nearest points from each class. These nearest points are called **support vectors** because they "support" (define) the margin.

Why Maximize the Margin?: - Think of the margin as a "no man's land" between classes - Larger margin □ more room for error on new data - Intuitively, a decision boundary right at the edge of your training data is fragile - A boundary with wide margins should generalize better

Hyperplane Basics: - A hyperplane in d dimensions: $H : \mathbf{w} \cdot \mathbf{x} + b = 0$ - **w** = normal vector (perpendicular to the hyperplane) - b = offset (distance from origin) - Points with $\mathbf{w} \cdot \mathbf{x} + b > 0$ are on one side; < 0 on the other

Distance from Point to Hyperplane:

$$d = \frac{|\mathbf{w} \cdot \mathbf{x} + b|}{||\mathbf{w}||}$$

This is just the projection of the point onto the normal vector, normalized by the length of **w**.

The Margin:

$$\gamma(\mathbf{w}, b) = \min_{x \in D} \frac{|\mathbf{w} \cdot \mathbf{x} + b|}{||\mathbf{w}||}$$

The margin is the distance to the *closest* point. It's scale-invariant: multiplying **w** and b by any constant doesn't change the hyperplane or its margin.

The Optimization Problem:

For binary classification with $y_i \in \{+1, -1\}$: - We need: $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) > 0$ for all points (correct classification) - We want: Maximize the margin

Original Formulation:

$$\max_{\mathbf{w},b} \min_{x \in D} \frac{|\mathbf{w} \cdot \mathbf{x} + b|}{||\mathbf{w}||} \quad \text{subject to} \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) > 0$$

This is a max-min problem—tricky to solve directly.

Clever Simplification: Since the hyperplane is scale-invariant, we can choose any scale. Let's fix:

$$|\mathbf{w} \cdot \mathbf{x} + b| = 1 \quad \text{for the points closest to the hyperplane}$$

Now the margin becomes $\frac{1}{||\mathbf{w}||}$, and maximizing margin = minimizing $||\mathbf{w}||$.

Final SVM Objective (Hard Margin):

$$\min \frac{1}{2}||\mathbf{w}||^2 \quad \text{subject to} \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i$$

Why $\frac{1}{2}||\mathbf{w}||^2$?: - Equivalent to minimizing $||\mathbf{w}||$ (squared is easier mathematically) - The $\frac{1}{2}$ makes derivatives cleaner

This is a Quadratic Programming (QP) problem: - Quadratic objective, linear constraints - Efficient solvers exist - Unique global solution (unlike perceptron, which finds any separating hyperplane)

Support Vectors: At the optimal solution, some points satisfy $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = 1$ exactly. These are the support vectors—they lie on the margin boundary and fully determine the solution.

Soft Margin SVM

The Problem: What if data isn't perfectly separable?

Hard margin SVM fails if: - Classes overlap - There are outliers - Data is noisy

The Solution: Allow some misclassifications, but penalize them.

Slack Variables (ξ_i): - Original constraint: $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$ - Relaxed constraint: $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i$ where $\xi_i \geq 0$

What ξ_i Represents: - $\xi_i = 0$: Point correctly classified and outside margin - $0 < \xi_i < 1$: Point correctly classified but inside margin - $\xi_i = 1$: Point exactly on the decision boundary - $\xi_i > 1$: Point misclassified

Soft Margin Objective:

$$\min \frac{1}{2}||\mathbf{w}||^2 + C \sum_i \xi_i \quad \text{subject to} \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

The Hinge Loss:

$$\xi_i = \max(1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b), 0)$$

This is the famous **hinge loss**—zero for points outside the margin, linear penalty for points inside or misclassified.

The C Parameter: - High C: Little tolerance for errors □ narrow margin, risk of overfitting - Low C: More tolerance for errors □ wide margin, risk of underfitting - $C = \infty$: Hard margin SVM

Duality

The Primal Problem (what we wrote above) can be hard to solve directly. Converting to the **dual problem** has advantages: - Often easier to solve - Reveals the kernel trick

Lagrangian Formulation: Introduce Lagrange multipliers α_i for each constraint:

$$L = \frac{1}{2}||\mathbf{w}||^2 - \sum_i \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1]$$

The Dual Problem (after taking derivatives):

$$\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j (\mathbf{x}_i^T \mathbf{x}_j)$$

$$\text{subject to} \quad \alpha_i \geq 0, \quad \sum_i \alpha_i y_i = 0$$

Key Insight: The data only appears as dot products $\mathbf{x}_i^T \mathbf{x}_j$!

Support Vectors and α_i: - If $\alpha_i = 0$: Point is not a support vector (doesn't affect solution) - If $\alpha_i > 0$: Point is a support vector

Most α_i are zero □ the solution depends only on support vectors.

Kernelization

The Problem: What if data isn't linearly separable in the original space?

The Solution: Map data to a higher-dimensional space where it might be linearly separable.

Feature Mapping: - Original data: **x** - Mapped data: $\phi(\mathbf{x})$ in higher (possibly infinite) dimension - Find a hyperplane in this new space

The Kernel Trick: Remember, the dual problem only uses dot products $\mathbf{x}_i^T \mathbf{x}_j$.

If we work in the mapped space, we need $\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$.

Key insight: We can define a **kernel function** $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ that computes this dot product *without ever explicitly computing* ϕ.

Common Kernels:

Polynomial Kernel:

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

Example: 1D points a and b, degree 2: - $K(a, b) = (ab + 1)^2 = a^2b^2 + 2ab + 1$ - This equals $\phi(a)^T\phi(b)$ where $\phi(x) = (x^2, \sqrt{2}x, 1)$

The kernel implicitly maps to a 3D space!

RBF (Gaussian) Kernel:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2\right)$$

Properties: - Similarity decreases exponentially with distance - Maps to **infinite-dimensional** space (via Taylor expansion of exponential) - Very flexible—can fit complex boundaries - γ controls the "reach" of each training point

Why Kernels are Powerful: 1. Compute high-dimensional dot products efficiently 2. Don't need to explicitly represent the high-dimensional vectors 3. Can work in infinite-dimensional spaces (RBF) 4. Turn linear methods into non-linear methods

SVM for Regression (SVR)

The Twist: Instead of maximizing margin between classes, we define a "tube" around the regression line and minimize points outside it.

ϵ-Insensitive Loss: - No penalty if prediction is within ϵ of true value - Linear penalty for points outside the tube

Formulation:

$$\min \frac{1}{2}||\mathbf{w}||^2 + C\sum_i(\xi_i + \xi_i^*)$$

Subject to: - $y_i - (\mathbf{w} \cdot \mathbf{x}_i + b) \leq \epsilon + \xi_i$ - $(\mathbf{w} \cdot \mathbf{x}_i + b) - y_i \leq \epsilon + \xi_i^*$ - $\xi_i, \xi_i^* \geq 0$

Intuition: - The regression line lies in the middle of the tube - Points inside the tube (within ϵ) don't contribute to the solution - Only points outside the tube become support vectors

Kernel Selection

Linear Kernel: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T\mathbf{x}_j$ - Fastest, most interpretable - Best when: Many features relative to samples (text classification) - No hyperparameters beyond C

Polynomial Kernel: $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T\mathbf{x}_j + c)^d$ - Captures feature interactions - Higher d = more complex boundaries - Parameters: degree d, coefficient c

RBF Kernel: $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2)$ - Most versatile, usually a good default - Higher γ = tighter fit around training points - Parameters: γ (often $\gamma = 1/(2\sigma^2)$)

Sigmoid Kernel: $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\alpha\mathbf{x}_i^T\mathbf{x}_j + c)$ - Similar to neural network activation - Less commonly used; can have convergence issues

Rule of Thumb: 1. Start with RBF (most flexible) 2. If that works well, try linear (faster, more interpretable) 3. Use cross-validation to select kernel and hyperparameters

SVM Hyperparameter Tuning

The C Parameter: - Trade-off: margin width vs. training accuracy - Low C: Wide margin, more misclassifications allowed, simpler model - High C: Narrow margin, few misclassifications, complex model (may overfit)

The γ Parameter (RBF kernel): - Controls influence radius of each support vector - Low γ: Large radius, smoother boundary, points influence far away - High γ: Small radius, complex boundary, points only influence locally

The Trade-off:

	Low C	High C
Low γ	Very smooth (underfit)	Smooth but fits training well
High γ	Wiggly but regularized	Very complex (overfit)

Tuning Strategy: 1. Grid search over C and γ on logarithmic scale 2. Example: $C \in \{0.001, 0.01, 0.1, 1, 10, 100, 1000\}$ 3. Example: $\gamma \in \{0.001, 0.01, 0.1, 1, 10\}$ 4. Use cross-validation to evaluate each combination 5. Select combination with best validation performance

Practical Tips: - Scale your features! SVMs are sensitive to feature scales - RBF kernel with well-tuned C and γ is often competitive with more complex methods - For large datasets, consider linear SVM (much faster) or approximations

ewpage

Dimensionality Reduction

High-dimensional data is everywhere: images (thousands of pixels), text (thousands of words), genomics (thousands of genes). Dimensionality reduction compresses this data into a smaller number of meaningful features while preserving important structure.

Background

The Curse of Dimensionality:

As dimensions increase, data becomes increasingly sparse. Consider: - A 1×1 square patch covers 1% of a 10×10 square - A 1×1×1 cubic patch covers 0.1% of a 10×10×10 cube - In general: Volume grows exponentially with dimension

Why This Matters: - Machine learning needs enough data to "fill" the space - Data requirements grow exponentially with dimensions - Distances become less meaningful (everything is "far" from everything) - Many algorithms break down

Two Approaches to Reduce Dimensions:

1. **Feature Selection**: Keep a subset of original features
   - Pros: Interpretable, simple
   - Cons: May lose important combinations
2. **Feature Extraction/Latent Features**: Create new features from combinations of originals
   - Linear methods: PCA, LDA
   - Non-linear methods: t-SNE, UMAP, autoencoders
   - Pros: Can capture more complex structure
   - Cons: New features may be hard to interpret

Principal Component Analysis (PCA)

PCA finds the directions of maximum variance in the data and projects onto them. It's the most widely used dimensionality reduction technique.

The Idea: - Find a linear projection from high dimension (D) to low dimension (L) - Preserve as much variance as possible - The directions of maximum variance are called **principal components**

Mathematical Setup: - Original data: $\mathbf{x} \in \mathbb{R}^D$ - Projection matrix: $\mathbf{W} \in \mathbb{R}^{D \times L}$ (columns are orthonormal) - **Encode**: $\mathbf{z} = \mathbf{W}^T\mathbf{x} \in \mathbb{R}^L$ - **Decode**: $\hat{\mathbf{x}} = \mathbf{W}\mathbf{z}$

Objective: Minimize reconstruction error

$$L(\mathbf{W}) = \frac{1}{N}\sum_i ||\mathbf{x}_i - \hat{\mathbf{x}}_i||^2$$

Derivation (projecting to 1D):

We want to find $\mathbf{w}_1$ that minimizes reconstruction error:

$$L = \frac{1}{N}\sum_i ||\mathbf{x}_i - z_{i1}\mathbf{w}_1||^2$$

Expanding and using $\mathbf{w}_1^T\mathbf{w}_1 = 1$ (orthonormality):

$$L = \frac{1}{N}\sum_i [\mathbf{x}_i^T\mathbf{x}_i - 2z_{i1}\mathbf{w}_1^T\mathbf{x}_i + z_{i1}^2]$$

Taking derivative w.r.t. z_{i1}:

$$\frac{\partial L}{\partial z_{i1}} = -2\mathbf{w}_1^T\mathbf{x}_i + 2z_{i1} = 0$$

Optimal encoding: $z_{i1} = \mathbf{w}_1^T\mathbf{x}_i$ (project onto $\mathbf{w}_1$)

Substituting back:

$$L = C - \frac{1}{N}\sum_i z_{i1}^2 = C - \frac{1}{N}\mathbf{w}_1^T\Sigma\mathbf{w}_1$$

Where Σ is the covariance matrix of **X**.

Key Insight: Minimizing reconstruction error = Maximizing variance of projections!

Using Lagrange multipliers with constraint $\mathbf{w}_1^T\mathbf{w}_1 = 1$:

$$\frac{\partial}{\partial \mathbf{w}_1}\left[\mathbf{w}_1^T\Sigma\mathbf{w}_1 + \lambda(1 - \mathbf{w}_1^T\mathbf{w}_1)\right] = 0$$

$$\Sigma\mathbf{w}_1 = \lambda\mathbf{w}_1$$

The optimal $\mathbf{w}_1$ is an eigenvector of the covariance matrix!

To maximize variance, choose the eigenvector with the **largest eigenvalue**.

Geometric Interpretation: - Imagine the data as a cloud of points - The first principal component is the direction of maximum spread - The second PC is perpendicular and captures the next most variance - And so on…

Think of it as finding the best “viewing angle” for your data.

Eigenvalues = Variance Explained: - Each eigenvalue equals the variance along that principal component - Sum of all eigenvalues = total variance - Fraction explained by first k components: $\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{j=1}^{D} \lambda_j}$

Scree Plot: Graph eigenvalues (or % variance) vs. component number - Look for an "elbow" where variance drops off - Components before the elbow are usually important

Factor Loadings: - Each PC is a linear combination of original features - Loadings = weights in this combination - High loading = feature contributes strongly to that PC

PCA + Regression: Still interpretable! - Run regression on principal components - Use loadings to translate back to original features

Computing PCA via SVD: - Singular Value Decomposition: $\mathbf{X} = \mathbf{USV}^T$ - **V** contains the principal components (eigenvectors) - **S** contains singular values (square roots of eigenvalues) - More numerically stable than eigendecomposition

Limitations of PCA: - Only captures **linear** relationships - Sensitive to **outliers** (they inflate variance) - Can't handle **missing data** (need imputation first) - **Unsupervised**: Doesn't use label information

Alternatives: - **Kernel PCA**: Non-linear version using the kernel trick - **Factor Analysis**: Assumes latent factors + noise - **LDA**: Supervised, maximizes between-class variance

Stochastic Neighbor Embedding (SNE)

The Idea: Preserve local neighborhood structure rather than global distances. Points that are neighbors in high-D should be neighbors in low-D.

Manifold Hypothesis: - High-dimensional data often lies on a lower-dimensional "manifold" - Think: Earth's surface is a 2D manifold in 3D space - We want to "unfold" this manifold

Algorithm:

1. **Convert distances to probabilities** (high-D):

$$p_{j|i} \propto \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma_i^2}\right)$$

This is the probability that point i would pick point j as its neighbor.

Adaptive variance (σ_i): - Dense regions get smaller σ_i (be more selective) - Sparse regions get larger σ_i (include more distant neighbors) - Controlled by **perplexity** parameter (roughly, the number of effective neighbors)

2. **Initialize low-D coordinates $\mathbf{z}_i$** randomly
3. **Compute low-D probabilities**:

$$q_{j|i} \propto \exp\left(-||\mathbf{z}_i - \mathbf{z}_j||^2\right)$$

4. **Minimize KL divergence** between p and q:

$$L = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

Why KL Divergence?: - If p is high but q is low: **Large penalty** (neighbors in high-D are far in low-D — bad!) - If p is low but q is high: **Small penalty** (distant points end up close — not as bad) - Prioritizes preserving local structure

Symmetric SNE: Make distances symmetric: $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2}$

t-SNE

The Problem with SNE: Crowding. Gaussian probability decays quickly, pushing moderately distant points too close together in low-D.

The Solution: Use the **t-distribution** (fat tails) for low-D probabilities:

$$q_{ij} \propto (1 + ||\mathbf{z}_i - \mathbf{z}_j||^2)^{-1}$$

Why t-Distribution?: - Heavier tails than Gaussian - Points that are moderately far in high-D can be *very* far in low-D - Creates well-separated clusters with tight internal structure

t-SNE Properties: - Excellent for visualization (2D or 3D) - Creates visually appealing, well-separated clusters - Preserves local structure well

Limitations: - **Computationally expensive**: $O(N^2)$ for pairwise distances - **Random initialization**: Results vary between runs - **Not invertible**: Can't go from low-D back to high-D - **Coordinates are meaningless**: Only relative positions matter - **Global structure distorted**: Distances between clusters are not meaningful

Hyperparameters: - **Perplexity**: Balance between local and global (typical: 5-50) - **Learning rate**: Step size for gradient descent - **Iterations**: Usually 1000+

UMAP

Uniform Manifold Approximation and Projection: Like t-SNE but faster and (arguably) better at preserving global structure.

Key Differences from t-SNE:

Aspect	t-SNE	UMAP
Pairwise distances	All pairs	Only neighbors
Initialization	Random	Spectral embedding
Updates	All points every iteration	Stochastic (subsets)
Speed	Slow ($O(N^2)$)	Much faster
Global structure	Often distorted	Better preserved

UMAP Algorithm:

1. **Build neighborhood graph** in high-D
    - Compute distance to k nearest neighbors
    - Convert to similarity scores (exponential decay from nearest neighbor)
2. **Make similarities symmetric**: $s_{ij} = s_{i|j} + s_{j|i} - s_{i|j} \cdot s_{j|i}$
3. **Initialize low-D with spectral embedding** (decomposition of graph Laplacian)
4. **Compute low-D similarities** using t-distribution variant:

$$q_{ij} \propto (1 + \alpha \cdot d^{2\beta})^{-1}$$

5. **Minimize cross-entropy** between high-D and low-D graphs

UMAP Advantages: - **Much faster** than t-SNE (especially for large datasets) - **Better global structure**: Preserves relative cluster distances better - `transform` **method**: Can embed new points without recomputing everything - **Flexible**: Can use any distance metric

When to Use Which: - **Small data, visualization only**: Either works, t-SNE often prettier - **Large data**: UMAP (t-SNE too slow) - **Need to embed new points**: UMAP - **Need reproducibility**: UMAP (more stable with same parameters)

Applications of Dimensionality Reduction

Data Visualization: - Reduce to 2D or 3D for plotting - Discover clusters, outliers, patterns visually - t-SNE and UMAP are standard tools

Noise Reduction: - Signal variance > noise variance - First few PCs capture signal, later PCs capture noise - Reconstructing from top PCs filters out noise

Preprocessing for ML: - Reduces curse of dimensionality - Speeds up training - Can improve performance (by removing noise) - Essential for algorithms sensitive to dimensionality (e.g., k-NN)

Feature Extraction: - Create more informative features - Example: Face recognition—first few PCs are “eigenfaces”

Multicollinearity: - Highly correlated predictors cause problems in regression - PCA creates uncorrelated components - Principal Component Regression (PCR) solves this

Comparing Techniques

Method	Linear?	Preserves	Best For	Interpretable?
PCA	Yes	Global variance	Preprocessing, compression	Yes (loadings)
t-SNE	No	Local neighborhoods	2D/3D visualization	No
UMAP	No	Local + some global	Large-scale visualization	No

Method	Linear?	Preserves	Best For	Interpretable?
LDA	Yes	Class separation	Supervised reduction	Yes

Selection Guide: 1. **Know you need visualization?** □ t-SNE or UMAP 2. **Need to preprocess for ML?** □ PCA 3. **Have class labels?** □ Consider LDA 4. **Need interpretability?** □ PCA 5. **Very large data?** □ UMAP or randomized PCA 6. **Complex non-linear structure?** □ UMAP, t-SNE, or kernel PCA

ewpage

Regression

Regression is the task of predicting a continuous outcome from input features. It's one of the oldest and most fundamental tools in statistics and machine learning, dating back to Gauss and Legendre in the early 1800s.

Bi-variate Regression

The Goal: Fit a straight line to data—understand how the response variable changes with one explanatory variable.

The Model:

$$E(y|x) = \hat{y} = a + bx$$

Where: - a = intercept (predicted y when $x = 0$) - b = slope (change in y for unit change in x)

Fitting the Line (Ordinary Least Squares):

Minimize the Sum of Squared Errors (SSE):

$$\text{SSE} = \sum_i (y_i - \hat{y}_i)^2$$

Solutions: - Slope: $b = \frac{S_{xy}}{S_{xx}} = \frac{\sum(x-\bar{x})(y-\bar{y})}{\sum(x-\bar{x})^2}$ - Intercept: $a = \bar{y} - b\bar{x}$

Why Squared Errors? - Penalizes large errors more than small ones - Mathematically convenient (differentiable) - Leads to closed-form solutions - Has nice statistical properties (BLUE under certain assumptions)

Important Concepts:

Outliers and Influential Points: - **Outlier**: Point far from the rest of the data - **Influential point**: Point that significantly affects the slope - A point can be an outlier without being influential (if x is near $\bar{x}$) - A point can be influential without being an outlier (high leverage)

Residual Variance (estimate of error variance):

$$s = \sqrt{\frac{\text{SSE}}{n-p}}$$

Where p = number of parameters (2 for simple regression).

We divide by $(n-p)$ not n because we "use up" degrees of freedom estimating parameters.

Homoscedasticity: Assumption that variance is constant across all x values.

Correlation:

$$r = \frac{\sum(x-\bar{x})(y-\bar{y})}{\sqrt{\sum(x-\bar{x})^2} \cdot \sqrt{\sum(y-\bar{y})^2}}$$

Properties: - Ranges from -1 to +1 - Measures strength of *linear* association - Relationship to slope: $r = \frac{s_x}{s_y} \cdot b$ - For standardized variables: $r = b$

Regression Toward the Mean: - Since $|r| \leq 1$, a 1 SD increase in x predicts less than 1 SD increase in y - Extreme values tend to be followed by less extreme values - This is why the technique is called "regression"!

R-Squared (Coefficient of Determination):

$$R^2 = \frac{\text{TSS} - \text{SSE}}{\text{TSS}} = 1 - \frac{\text{SSE}}{\text{TSS}}$$

Where: - TSS = Total Sum of Squares = $\sum(y-\bar{y})^2$ (variance in y) - SSE = Sum of Squared Errors = $\sum(y-\hat{y})^2$ (unexplained variance)

Interpretation: Proportion of variance in y explained by x.

For simple regression: $R^2 = r^2$ (squared correlation)

Statistical Significance (Is the slope different from zero?):

$$t = \frac{b}{\text{SE}(b)} = \frac{b}{s/\sqrt{S_{xx}}}$$

This follows a t-distribution with $(n-2)$ degrees of freedom.

Equivalently, the F-test: $F = t^2 = \frac{R^2/1}{(1-R^2)/(n-2)}$

Multivariate Regression

The Model:

$$E(y|x) = \hat{y} = a + b_1x_1 + b_2x_2 + ... + b_px_p$$

Interpreting Coefficients: - b_1 is the effect of x_1 on y **holding all other variables constant** - This is called the **partial regression coefficient** - Very different from simple regression coefficient!

Why Controlling Matters (Types of Relationships):

Relationship	What Happens
Confounding	Third variable causes both x and y; controlling reveals true (weaker) relationship
Mediation	Third variable transmits effect from x to y; controlling removes indirect effect
Suppression	Third variable masks relationship; controlling reveals hidden relationship

Partial Regression Plots: Visualize the true relationship between x_1 and y after removing the effect of other variables:

1. Regress y on all variables except x_1 □ get residuals e_y
2. Regress x_1 on all other x variables □ get residuals e_{x_1}
3. Plot e_y vs e_{x_1}

The slope of this plot equals b_1 from the full model.

Statistical Tests:

F-test (Are any predictors significant?):

$$F = \frac{R^2/(p-1)}{(1-R^2)/(n-p)}$$

Tests whether the model explains more variance than expected by chance.

t-test (Is a specific predictor significant?):

$$t = \frac{b_j}{\text{SE}(b_j)}$$

Tests whether b_j is significantly different from zero.

Comparing Nested Models: - Complete model: All variables - Reduced model: Some variables dropped

$$F = \frac{(\text{SSE}_r - \text{SSE}_c)/(df_c - df_r)}{\text{SSE}_c/df_c}$$

ANOVA Table (Partitioning Variance):

Source	Sum of Squares	df	Mean Square
Regression	$\sum(\hat{y} - \bar{y})^2$	$p-1$	SSR/(p-1)
Error	$\sum(y - \hat{y})^2$	$n-p$	SSE/(n-p)
Total	$\sum(y - \bar{y})^2$	$n-1$	—

$F = \text{MSR}/\text{MSE}$

Bonferroni Correction: When testing multiple coefficients, divide significance level by number of tests to control overall Type I error.

Logistic Regression

The Problem: Linear regression for binary outcomes predicts values outside [0,1].

The Solution: Model the probability using a sigmoid (S-shaped) curve.

The Model:

$$P(y = 1|x) = \frac{e^{\alpha+\beta x}}{1 + e^{\alpha+\beta x}} = \frac{1}{1 + e^{-(\alpha+\beta x)}}$$

Equivalently, the **log-odds** (logit) is linear:

$$\log\left(\frac{P(y = 1)}{1 - P(y = 1)}\right) = \alpha + \beta x$$

Why Log-Odds? - Odds can range from 0 to ∞ - Log-odds can range from -∞ to +∞ - Makes sense to model with a linear function

Interpreting Coefficients: - β = change in log-odds for unit increase in x - e^{β} = **odds ratio** for unit increase in x - If $\beta = 0.5$, then $e^{0.5} \approx 1.65$: the odds multiply by 1.65 for each unit of x

Propensity Scores (Causal Inference):

When comparing treatment groups, selection bias can confound results.

Propensity score = $P(\text{treatment}|\text{covariates})$

Use logistic regression to estimate propensity, then: 1. Match treated/control units with similar propensity 2. Weight by inverse propensity 3. Stratify by propensity quintiles

This “balances” groups on observed covariates.

Model Comparison:

Likelihood Ratio Test:

$$\chi^2 = -2(\log L_{\text{reduced}} - \log L_{\text{full}})$$

Follows chi-squared distribution with df = difference in number of parameters.

Wald Test: $(b_j/\text{SE}(b_j))^2$ follows chi-squared(1).

Ordinal Logistic Regression (ordered categories): - Model cumulative probabilities: $P(y \leq j)$ - Same slopes across all cutpoints (proportional odds assumption)

Multinomial Logistic Regression (unordered categories): - One-vs-Rest: Separate model for each class - One-vs-One: Model for each pair of classes

Regression Diagnostics

Good regression analysis requires checking assumptions.

Residual Analysis: - **Residuals vs. Fitted**: Should show no pattern (random scatter around 0) - **Q-Q Plot**: Residuals should follow the diagonal (normality check) - **Scale-Location**: Spread should be constant (homoscedasticity check) - **Residuals vs. Leverage**: Identifies influential outliers

Multicollinearity (correlated predictors): - Makes coefficient estimates unstable - Inflates standard errors

Variance Inflation Factor (VIF):

$$\text{VIF}_j = \frac{1}{1 - R_j^2}$$

Where R_j^2 is from regressing x_j on all other predictors.

Interpretation: - VIF = 1: No correlation with other predictors - VIF = 5: Moderate multicollinearity - VIF > 10: Serious problem—consider removing variables or using regularization

Influential Observations:

Measure	What It Detects
Leverage (hat values)	Points far from $\bar{x}$ that *could* influence fit
Residual	Points far from fitted line
Cook's Distance	Combined influence on all predictions
DFBETA	Effect on individual coefficient estimates

High leverage + large residual = influential point.

Model Selection Criteria: - **AIC** = $2k - 2\ln(L)$: Penalizes complexity (lower is better) - **BIC** = $k\ln(n) - 2\ln(L)$: Stronger penalty, favors simpler models - **Adjusted** R^2: R^2 penalized for number of predictors

Where k = number of parameters, L = likelihood, n = sample size.

Advanced Regression Techniques

Ridge Regression (L2 regularization):

$$\min_{\beta} ||\mathbf{y} - \mathbf{X}\beta||^2 + \lambda||\beta||^2$$

Properties: - Shrinks coefficients toward zero (but never exactly zero) - Reduces variance at cost of some bias - Excellent for multicollinearity - All predictors kept in model

Lasso Regression (L1 regularization):

$$\min_{\beta} ||\mathbf{y} - \mathbf{X}\beta||^2 + \lambda||\beta||_1$$

Properties: - Shrinks some coefficients exactly to zero - Performs automatic **feature selection** - Great for high-dimensional, sparse problems - Selects only one from a group of correlated predictors

Elastic Net (L1 + L2):

$$\min_{\beta} ||\mathbf{y} - \mathbf{X}\beta||^2 + \lambda_1||\beta||_1 + \lambda_2||\beta||^2$$

Properties: - Combines benefits of Ridge and Lasso - Can select groups of correlated features - Two hyperparameters to tune

Choosing λ: Use cross-validation to find the value that minimizes prediction error on held-out data.

Quantile Regression: - Standard regression models the **mean**: $E(y|x)$ - Quantile regression models **quantiles**: e.g., median, 10th percentile - Robust to outliers - Shows how *distribution* of y changes with x

When to Use: - When relationship differs across the distribution (e.g., effect on high vs. low income) - When outliers are a concern - When you care about specific quantiles (e.g., 95th percentile for risk)

ewpage

Part II: Elements of Statistical Learning Notes

ewpage

Introduction to Statistical Learning

These notes are based on “The Elements of Statistical Learning” (ESL) by Hastie, Tibshirani, and Friedman — one of the foundational textbooks in machine learning and statistical modeling. This guide is designed to make these concepts accessible to undergraduates while maintaining the mathematical depth needed for a solid understanding.

What is Statistical Learning?

Statistical learning refers to a set of tools for understanding data. These tools can be broadly classified into two categories:

- **Supervised Learning**: We have input variables (features) and an output variable (response), and we want to learn the relationship between them. Examples include predicting house prices from features like square footage and location (regression), or classifying emails as spam or not spam (classification).
- **Unsupervised Learning**: We only have input variables and want to discover patterns or structure in the data. Examples include grouping customers into segments (clustering) or reducing the dimensionality of data (PCA).

Why This Book Matters

ESL bridges the gap between statistical theory and practical machine learning. Understanding these foundations helps you:

1. **Choose the right model** for your problem
2. **Understand trade-offs** (like bias vs. variance)
3. **Avoid common pitfalls** (like overfitting)
4. **Interpret results** correctly

Topics Covered

Chapter	Topic	What You’ll Learn
3	Linear Regression	How to model linear relationships and interpret coefficients

Chapter	Topic	What You'll Learn
4	Linear Classification	How to classify data into categories using linear boundaries
6	Kernel Methods	How to capture non-linear patterns using kernel functions
7	Model Assessment and Selection	How to evaluate models and choose the best one
8	Model Inference and Averaging	How to quantify uncertainty and combine models
9	Additive Models, Trees	How to build interpretable non-linear models
10	Boosting and Additive Trees	How to combine weak learners into powerful predictors
15	Random Forests	How to build robust ensemble models

Prerequisites

To get the most out of these notes, you should be comfortable with:

- **Calculus**: Derivatives, gradients, and optimization
- **Linear Algebra**: Matrices, vectors, eigenvalues
- **Probability & Statistics**: Distributions, expectations, variance, hypothesis testing
- **Basic Programming**: For implementing these algorithms

How to Use These Notes

1. **Read the intuitive explanations first** — understand the "why" before the "how"
2. **Work through the math slowly** — the equations encode important insights
3. **Connect concepts to real examples** — think about where each method applies
4. **Practice on datasets** — implementation solidifies understanding

Let's begin!

ewpage

Linear Regression

Linear regression is one of the most fundamental tools in statistics and machine learning. It models the relationship between a continuous response variable and one or more predictor variables. Despite its simplicity, it forms the foundation for understanding more complex methods.

The Big Picture

Imagine you want to predict house prices based on features like square footage, number of bedrooms, and location. Linear regression assumes that the price is approximately a weighted sum of these features, plus some random noise.

Key Insight: Linear regression finds the "best" weights (coefficients) that minimize the difference between predicted and actual values.

Model Formulation

The Linear Model

We assume the response variable Y relates to predictors X through:

$$y = X\beta + \epsilon$$

Where: - **y**: Vector of observed outcomes (e.g., house prices) - **X**: Matrix of predictor values (each row is one observation, each column is one feature) - **β**: Coefficients we want to estimate (the "weights") - **ε**: Random error term, assumed to follow $\epsilon \sim N(0, \sigma^2 I)$

The assumption of Gaussian errors is important — it means errors are symmetric around zero, with most errors being small and large errors being rare.

Linear Function Approximation

We're approximating the conditional expectation:

$$E(Y|X) = f(X) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j$$

Interpretation of coefficients: β_j represents the expected change in Y for a one-unit increase in x_j, *holding all other predictors constant*. This "holding constant" interpretation is crucial and often misunderstood!

Note: The model is "linear" in the *parameters* (β), not necessarily in the predictors. You can include x^2, log(x), or interactions — the model is still "linear" because coefficients enter linearly.

Finding the Best Coefficients

Least Squares: The Objective

We want coefficients that make our predictions as close as possible to the actual values. We measure "closeness" using the Residual Sum of Squares (RSS):

$$RSS = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{N}(y_i - f(x_i))^2 = (y - X\beta)^T(y - X\beta)$$

Why squared errors? Squaring penalizes large errors more heavily, gives a smooth (differentiable) objective function, and leads to elegant closed-form solutions.

Deriving the Solution

To minimize RSS, we take the derivative with respect to β and set it to zero:

$$\frac{\partial RSS}{\partial \beta} = -2X^T(y - X\beta) = 0$$

Solving for β gives us the famous **Normal Equations**:

$$\hat{\beta} = (X^TX)^{-1}X^Ty$$

This is the Ordinary Least Squares (OLS) estimator.

Predicted Values and the Hat Matrix

The fitted values are:

$$\hat{y} = X\hat{\beta} = X(X^TX)^{-1}X^Ty = Hy$$

Where **H** is called the **Hat Matrix** or projection matrix: - H "puts the hat" on y (transforms y into $\hat{y}$) - The diagonal elements H_{ii} are called **leverage values** - High leverage points have outsized influence on the fit - Leverage values range from 1/N to 1, with average p/N

Understanding Uncertainty

Sampling Distribution of β

If we collected new data and re-estimated β, we'd get slightly different values. The sampling distribution tells us about this variability:

$$\hat{\beta} \sim N(\beta, (X^T X)^{-1}\sigma^2)$$

What this means: Our estimated coefficients follow a normal distribution centered on the true coefficients (unbiased!), with a variance that depends on: - The noise level σ² (more noise = more uncertainty) - The structure of the predictors $(X^T X)^{-1}$

Estimating the Noise Level

Since σ² is unknown, we estimate it from the data:

$$\hat{\sigma}^2 = \frac{1}{N-p-1}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \frac{RSS}{N-p-1}$$

Why N-p-1? We lose one degree of freedom for each parameter we estimate (p slopes + 1 intercept), leaving N-p-1 degrees of freedom.

Testing Statistical Significance

Is a Coefficient Different from Zero?

For each coefficient, we can test whether it's significantly different from zero using a t-test:

$$Z_j = \frac{\hat{\beta}_j}{\text{SE}(\hat{\beta}_j)} = \frac{\hat{\beta}_j}{\hat{\sigma}\sqrt{v_j}}$$

Where v_j is the j-th diagonal element of $(X^T X)^{-1}$.

Under the null hypothesis $H_0 : \beta_j = 0$, this statistic follows a t-distribution with N-p-1 degrees of freedom.

Practical interpretation: A large |Z| (typically > 2) suggests the coefficient is statistically significant, meaning we have evidence that the predictor matters for predicting Y.

Testing Groups of Coefficients

Sometimes we want to test whether a group of coefficients (e.g., all levels of a categorical variable) are jointly zero. We use the F-test:

$$F = \frac{(RSS_0 - RSS_1)/(p_1 - p_0)}{RSS_1/(N - p_1 - 1)}$$

Where: - RSS_0 = RSS from the restricted model (without the group) - RSS_1 = RSS from the full model (with the group) - p_0, p_1 = number of parameters in each model

Under the null hypothesis, $F \sim F_{p_1-p_0, N-p_1-1}$.

Gauss-Markov Theorem

This is one of the most important theoretical results in linear regression:

Statement: Among all *linear, unbiased* estimators, OLS has the *smallest variance*.

This is why OLS is called **BLUE** (Best Linear Unbiased Estimator).

Expected Prediction Error

When predicting a new observation:

$$E[(Y_0 - \hat{Y}_0)^2] = \sigma^2 + \text{MSE}(\hat{f}(X_0))$$

The prediction error has two components: 1. **Irreducible error** (σ^2): randomness we can't eliminate 2. **Estimation error** (MSE): uncertainty from estimating f

Assumptions Required

The Gauss-Markov theorem relies on these assumptions: 1. **Linearity**: The true relationship is linear 2. **Independence**: Errors are independent across observations 3. **Homoscedasticity**: Error variance is constant (not changing with X) 4. **No perfect multicollinearity**: Predictors aren't perfectly correlated

Important: Gauss-Markov says OLS is best among unbiased estimators. But sometimes a *biased* estimator with lower variance gives better predictions (see Shrinkage Methods below).

Subset Selection

When you have many predictors, some may be irrelevant. Including them adds noise and hurts interpretability. Subset selection methods help identify the most important predictors.

Best Subset Selection

Idea: For each subset size k, find the k variables that minimize RSS. Choose the best k using cross-validation or information criteria.

Problem: With p predictors, there are 2^p possible subsets — computationally infeasible for large p.

Forward Selection

Start with no predictors and iteratively add the one that most improves the fit:

1. Start with intercept only
2. For each remaining predictor, compute RSS if added
3. Add the predictor that reduces RSS the most
4. Repeat until stopping criterion (e.g., no significant improvement)

Pros: Computationally efficient (O(p²) instead of O(2^p)) **Cons**: May miss the optimal subset (greedy algorithm)

Technical note: QR decomposition or successive orthogonalization can efficiently compute which variable to add next.

Backward Selection

Start with all predictors and iteratively remove the least important:

1. Start with all p predictors
2. Compute the t-statistic for each coefficient
3. Remove the predictor with smallest |t| (least significant)
4. Repeat until stopping criterion

Pros: Considers all variables initially, captures interactions **Cons**: Requires N > p (can't start with all variables if you have more variables than observations)

Hybrid Stepwise Selection

At each step, consider both adding and removing variables. Use criteria like AIC to guide decisions. This explores the model space more thoroughly than pure forward or backward selection.

Forward Stagewise Selection

A more gradual approach: 1. Start with all coefficients at zero 2. Find the variable most correlated with the current residuals 3. Add a *small* amount of that variable's coefficient (don't fully optimize) 4. Repeat

This is similar to gradient descent in function space and connects to boosting methods discussed later.

Shrinkage Methods

Subset selection is "all or nothing" — a variable is either in or out. Shrinkage methods take a softer approach: they keep all variables but *shrink* coefficients toward zero.

Key insight: Accepting a small amount of bias in exchange for reduced variance often improves prediction accuracy.

Ridge Regression

Ridge regression adds a penalty on the size of coefficients:

$$\hat{\beta}^{\text{ridge}} = \arg\min_{\beta} \left[(y - X\beta)^T (y - X\beta) + \lambda \sum_{j=1}^{p} \beta_j^2 \right]$$

Equivalently, it's a constrained optimization:

$$\hat{\beta}^{\text{ridge}} = \arg\min_{\beta} (y - X\beta)^T (y - X\beta) \quad \text{subject to} \quad \sum \beta_j^2 \leq t$$

Here: - **λ** (or equivalently, **t**) controls the strength of the penalty - λ = 0 gives ordinary OLS - λ □ ∞ shrinks all coefficients toward zero

Closed-form solution:

$$\hat{\beta}^{\text{ridge}} = (X^T X + \lambda I)^{-1} X^T y$$

Why does Ridge help with correlated predictors?

When predictors are correlated, $(X^T X)$ is nearly singular (not full rank). This makes OLS coefficients unstable — small changes in data cause huge changes in coefficients. Adding λI to the diagonal "regularizes" the matrix, making inversion stable.

Geometric interpretation using eigenvalue decomposition:

$X^T X = UDU^T$ where D is diagonal with eigenvalues $d_1, ..., d_p$

Ridge coefficients become: $\hat{\beta}^{\text{ridge}} = \sum_{j=1}^{p} \frac{d_j}{d_j + \lambda} u_j^T y \cdot u_j$

The term $\frac{d_j}{d_j + \lambda}$ shrinks coefficients most in directions with small eigenvalues (high collinearity).

Important: Ridge regression is NOT scale invariant — you must standardize predictors before applying it. Don't penalize the intercept.

Lasso Regression

Lasso uses an L1 penalty instead of L2:

$$\hat{\beta}^{\text{lasso}} = \arg\min_{\beta} \left[(y - X\beta)^T (y - X\beta) + \lambda \sum_{j=1}^{p} |\beta_j| \right]$$

Key difference from Ridge: Lasso can shrink coefficients exactly to zero, performing automatic variable selection!

Why does L1 give sparsity?

Geometrically: - Ridge constraint region: a disk/sphere ($\beta_1^2 + \beta_2^2 \leq t$) - Lasso constraint region: a diamond/rhombus ($|\beta_1| + |\beta_2| \leq t$)

The diamond has corners on the axes. The optimal solution often hits a corner, setting some coefficients exactly to zero.

Bayesian interpretation: - Lasso = MAP estimation with Laplace prior: $p(\beta) \propto e^{-\alpha|\beta|}$ - Ridge = MAP estimation with Gaussian prior: $p(\beta) \propto e^{-\alpha\beta^2/2}$

The Laplace prior has heavier tails and more mass at zero, encouraging sparsity.

Computation: Unlike Ridge, Lasso requires iterative optimization (coordinate descent is popular).

Elastic Net

Elastic Net combines L1 and L2 penalties:

$$\text{Penalty} = \lambda \left[\alpha \sum |\beta_j| + (1 - \alpha) \sum \beta_j^2 \right]$$

Benefits: - Variable selection like Lasso (from L1) - Stability with correlated predictors like Ridge (from L2) - Better handles groups of correlated predictors (selects groups together)

Partial Least Squares (PLS)

When predictors are highly correlated, both OLS and Ridge can struggle. PLS offers an alternative approach by constructing new features that capture both high variance AND high correlation with the response.

Comparing to Principal Component Regression (PCR)

- **PCR**: First does PCA on X (ignoring Y), then regresses Y on the top principal components
- **PLS**: Finds directions in X that have both high variance AND high correlation with Y (supervised)

PLS Algorithm

1. Standardize X and y (zero mean, unit variance)
2. For m = 1, 2, …, M components:
   - Compute weight vector: $w_m \propto X_{m-1}^T y$ (direction most correlated with response)
   - Create score: $z_m = X_{m-1} w_m$
   - Regress y on z_m to get coefficient $\hat{\phi}_m$
   - Regress each column of X_{m-1} on z_m to get loadings $\hat{p}_m$
   - Orthogonalize: $X_m = X_{m-1} - z_m \hat{p}_m^T$ (remove what's explained)
3. Final prediction: $\hat{y} = \bar{y} + \sum_{m=1}^{M} \hat{\phi}_m z_m$

When to use PLS: High-dimensional data with many correlated predictors (common in chemometrics, genomics).

Summary: Choosing a Method

Method	Best For	Pros	Cons
OLS	Low-dimensional, well-behaved data	Unbiased, interpretable	Poor with collinearity or p > N
Ridge	Collinear predictors	Stable, works when p > N	Keeps all variables
Lasso	Sparse signals	Automatic variable selection	Can be unstable with correlated predictors
Elastic Net	Correlated groups of predictors	Best of both worlds	Extra tuning parameter
PLS	High-dimensional, many correlated predictors	Dimension reduction + prediction	Less interpretable

ewpage

Classification

Classification is one of the most common machine learning tasks. Unlike regression (where we predict a continuous number), in classification we predict which *category* or *class* an observation belongs to. Examples include: spam vs. not spam, disease vs. healthy, or recognizing handwritten digits (0-9).

The Big Picture

In classification, we want to: 1. **Learn decision boundaries** that separate different classes 2. **Estimate probabilities** of class membership 3. **Make predictions** for new observations

The methods in this chapter produce *linear* decision boundaries — the simplest and most interpretable kind.

Decision Boundaries

What is a Decision Boundary?

A decision boundary is the dividing line (or surface, in higher dimensions) between regions assigned to different classes. When a new observation falls on one side, we predict one class; on the other side, we predict the other class.

Linear Decision Boundaries

The boundary is linear if it's described by a linear equation in the features:

$$\{x : w_0 + w_1 x_1 + w_2 x_2 + ... + w_p x_p = 0\}$$

In 2D, this is a straight line. In 3D, it's a plane. In higher dimensions, it's a **hyperplane**.

How Do We Get Linear Boundaries?

The decision boundary is linear if any of these conditions hold: - The discriminant function $\delta_k(x)$ is linear in x - The posterior probability $P(G = k|X = x)$ is linear in x - Some monotonic transformation of the above is linear

Discriminant Functions

We learn a **discriminant function** $\delta_k(x)$ for each class k. To classify a new point x: - Compute $\delta_k(x)$ for all classes - Assign x to the class with the highest discriminant value

For linear discriminant functions:

$$\delta_k(x) = \beta_{0k} + \beta_k^T x$$

The decision boundary between classes k and l is where $\delta_k(x) = \delta_l(x)$:

$$\{x : (\beta_{0k} - \beta_{0l}) + (\beta_k - \beta_l)^T x = 0\}$$

This is clearly linear in x — an affine set (hyperplane not necessarily through the origin).

Example: Logistic Regression Boundary

Binary logistic regression models probabilities as:

$$P(G = 1|X = x) = \frac{\exp(\beta^T x)}{1 + \exp(\beta^T x)} = \frac{1}{1 + \exp(-\beta^T x)}$$

$$P(G = 0|X = x) = \frac{1}{1 + \exp(\beta^T x)}$$

The **log-odds** (also called logit) is:

$$\log\left(\frac{P(G = 1|x)}{P(G = 0|x)}\right) = \beta^T x$$

This is linear in x! So the decision boundary — where both classes are equally likely — is:

$$\{x : \beta^T x = 0\}$$

Linear Probability Model (and Why It Fails)

The Approach

One simple idea: encode classes as numbers (0/1 for binary, or indicator matrix Y for multiclass) and just run linear regression!

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$
$$\hat{Y} = X\hat{\beta}$$

Problems with This Approach

1. **Predictions outside [0,1]**: Linear regression can predict negative probabilities or probabilities greater than 1 — nonsense for probabilities!
2. **Class masking**: With multiple classes and few features, one class can be "dominated" everywhere. Imagine three classes where class 2 always has lower predicted values than classes 1 or 3 — the model will never predict class 2!

Takeaway: Use methods designed for classification, not regression hacks.

Linear Discriminant Analysis (LDA)

LDA is one of the oldest and most elegant classification methods. It takes a generative approach: model how the data is *generated* for each class, then use Bayes' theorem to classify.

The Generative Model

Assume that within each class k, the data follows a multivariate normal distribution:

$$X|G = k \sim N(\mu_k, \Sigma)$$

Key assumption for LDA: All classes share the same covariance matrix Σ (but have different means μ_k).

Using Bayes' Theorem

Once we model how data is generated, we can compute:

$$P(G = k|X = x) = \frac{f_k(x) \times \pi_k}{\sum_l f_l(x) \times \pi_l}$$

Where: - π_k = prior probability of class k (how common is this class?) - $f_k(x)$ = class-conditional density (how likely is x given class k?)

The Discriminant Function

For a multivariate normal:

$$f_k(x) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k)\right\}$$

Taking the log and simplifying (using the common Σ assumption), we get the **linear discriminant function**:

$$\delta_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log \pi_k$$

This is linear in x, hence “Linear” Discriminant Analysis.

Decision Boundary

The log-odds between classes k and l is:

$$\log \frac{P(G=k|x)}{P(G=l|x)} = C + x^T\Sigma^{-1}(\mu_k - \mu_l)$$

Linear in x! The constant terms combine nicely because Σ is shared across classes.

Estimation in Practice

We estimate the parameters from training data: - **Prior**: $\hat{\pi}_k = N_k/N$ (proportion of each class) - **Mean**: $\hat{\mu}_k = \frac{1}{N_k}\sum_{i\in\text{class } k} x_i$ (class centroid) - **Covariance**: $\hat{\Sigma} = \frac{1}{N}\sum_k \sum_{i\in\text{class } k}(x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T$ (pooled covariance)

Quadratic Discriminant Analysis (QDA)

Relaxing the Equal Covariance Assumption

What if each class has its own covariance structure? QDA allows:

$$X|G=k \sim N(\mu_k, \Sigma_k)$$

Quadratic Discriminant Function

Now the discriminant becomes:

$$\delta_k(x) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) + \log \pi_k$$

This is **quadratic** in x — the decision boundaries are curved (ellipses, parabolas, hyperbolas).

Trade-offs: LDA vs QDA

Aspect	LDA	QDA
Boundaries	Linear	Quadratic (curved)
Parameters	~Kp means + p(p+1)/2 covariance	~Kp means + Kp(p+1)/2 covariances
Flexibility	Less flexible	More flexible
Variance	Lower (fewer parameters)	Higher (more parameters)
Best when	Classes have similar spread	Classes have different shapes/orientations

Regularized Discriminant Analysis

A compromise between LDA and QDA:

$$\hat{\Sigma}_k(\alpha) = \alpha\Sigma_k + (1-\alpha)\Sigma$$

- α = 0: LDA (shared covariance)
- α = 1: QDA (class-specific covariance)
- 0 < α < 1: Smooth interpolation

Choose α via cross-validation.

Naive Bayes Classifier

The Independence Assumption

Naive Bayes makes a strong (and usually wrong!) assumption: given the class, all features are **conditionally independent**:

$$f_k(x) = \prod_{j=1}^{p} f_{kj}(x_j)$$

Why "Naive"?

This assumption rarely holds in practice. If predicting whether an email is spam, the presence of "free" and "money" are probably correlated. But Naive Bayes ignores this.

Why Does It Still Work?

Despite the wrong assumption, Naive Bayes often works well because: 1. We only need to get the *ranking* of classes right, not exact probabilities 2. The errors from the independence assumption may cancel out 3. It's extremely fast and scales to high dimensions

The Log-Odds

$$\log \frac{P(G=k|x)}{P(G=l|x)} = \log \frac{\pi_k}{\pi_l} + \sum_{j=1}^{p} \log \frac{f_{kj}(x_j)}{f_{lj}(x_j)}$$

Each feature contributes independently to the log-odds — simple and interpretable!

Logistic Regression

Logistic regression is the workhorse of classification. Unlike LDA (which models P(XIG)), logistic regression directly models P(GIX).

The Model

For binary classification:

$$P(G=1|X=x) = \frac{\exp(\beta^T x)}{1+\exp(\beta^T x)} = \sigma(\beta^T x)$$

$$P(G=0|X=x) = \frac{1}{1+\exp(\beta^T x)} = 1-\sigma(\beta^T x)$$

Where σ(·) is the **sigmoid function** — it squashes any real number into (0,1).

Why the Sigmoid?

The sigmoid ensures probabilities are always between 0 and 1, and the log-odds is linear:

$$\log \frac{P(G=1|x)}{P(G=0|x)} = \beta^T x$$

Maximum Likelihood Estimation

We find β by maximizing the probability of the observed labels. The log-likelihood is:

$$\ell(\beta) = \sum_{i=1}^{N} \left[y_i \log p(x_i,\beta) + (1-y_i) \log(1-p(x_i,\beta)) \right]$$

Or equivalently:

$$\ell(\beta) = \sum_{i=1}^{N} \left[y_i(\beta^T x_i) - \log(1 + \exp(\beta^T x_i)) \right]$$

Finding the Maximum

Taking the derivative (the **score function**):

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{N} x_i(y_i - p(x_i, \beta)) = X^T(y - p)$$

Setting this to zero: the predicted probabilities should "balance" with the actual labels.

Optimization: Newton-Raphson / IRLS

The log-likelihood is non-linear in β — no closed-form solution. We use iterative methods.

The **Hessian** (second derivative) is:

$$\frac{\partial^2 \ell}{\partial \beta \partial \beta^T} = -\sum_{i=1}^{N} x_i x_i^T p(x_i, \beta)(1 - p(x_i, \beta)) = -X^T W X$$

Where W is diagonal with $W_{ii} = p_i(1 - p_i)$.

Good news: The Hessian is negative definite, so the log-likelihood is **concave** — there's a unique global maximum!

The algorithm is called **Iteratively Reweighted Least Squares (IRLS)** because each iteration looks like a weighted least squares problem.

Measuring Goodness of Fit: Deviance

Deviance measures how far our model is from a "perfect" model:

$$\text{Deviance} = -2(\log L_M - \log L_S)$$

Where: - L_M = likelihood of our model - L_S = likelihood of the saturated model (perfect fit)

To compare models, look at the change in deviance (larger drops = better improvement).

Regularization

Just like in regression, we can add penalties: - **L2 penalty** (Ridge): Shrinks coefficients, helps with correlated predictors - **L1 penalty** (Lasso): Shrinks some coefficients to exactly zero

$$\text{Maximize: } \ell(\beta) - \lambda \sum_{j=1}^{p} |\beta_j|$$

Note: Don't penalize the intercept!

LDA vs. Logistic Regression

Both produce linear decision boundaries, but they differ in important ways:

Aspect	LDA	Logistic Regression
Approach	Generative (models P(XIG))	Discriminative (models P(GIX))
Likelihood	Full joint likelihood	Conditional likelihood
Assumptions	Normal class distributions, equal covariances	Just that log-odds is linear
Efficiency	More efficient if assumptions hold	More robust to violations
Outliers	Sensitive (Gaussian assumption)	More robust
When to use	Small samples, well-behaved data	Large samples, suspect non-normality

Rule of thumb: If you have small samples and trust the Gaussian assumption, LDA can be more efficient. Otherwise, logistic regression is safer.

Perceptron Learning Algorithm

The perceptron is a historically important algorithm (precursor to neural networks) that finds a separating hyperplane.

The Objective

Minimize the total distance of misclassified points to the decision boundary:

$$D(\beta) = -\sum_{i \in \text{misclassified}} y_i(x_i^T\beta)$$

Algorithm

Use stochastic gradient descent: 1. Initialize β 2. For each misclassified point i: - Update: $\beta \leftarrow \beta + \eta \cdot y_i \cdot x_i$ 3. Repeat until convergence

Limitations

- **Multiple solutions**: If data is separable, many hyperplanes work. The final answer depends on initialization and order of updates.
- **No convergence guarantee**: If data is NOT separable, the algorithm never converges — it just bounces around forever.

This motivates the **Support Vector Machine**, which finds the unique "best" separating hyperplane.

Maximum Margin Classifiers

The Margin Idea

If data is linearly separable, many hyperplanes separate the classes. Which is "best"?

Intuition: Choose the hyperplane with the largest **margin** — the distance from the hyperplane to the nearest points of either class. This gives the most "room for error" on new data.

Mathematical Formulation

We want to maximize the margin M:

$$\max_{\beta, \|\beta\|=1} M \quad \text{subject to} \quad y_i(x_i^T\beta) \geq M \quad \forall i$$

Reformulation (dropping the norm constraint):

$$\min_{\beta} \frac{1}{2}\|\beta\|^2 \quad \text{subject to} \quad y_i(x_i^T\beta) \geq 1 \quad \forall i$$

This is a **quadratic program** — convex optimization with a unique solution.

Lagrangian Formulation

$$L = \frac{1}{2}\|\beta\|^2 - \sum_{i=1}^{N} \alpha_i \left[y_i(x_i^T\beta) - 1\right]$$

Taking the derivative with respect to β:

$$\beta = \sum_{i=1}^{N} \alpha_i y_i x_i$$

Key insight: The solution is a linear combination of the training points! But only points where the constraint is active (on the margin boundary) contribute — these are called **support vectors**.

For most points, $\alpha_i = 0$. Only a few points determine the decision boundary. This makes SVMs efficient and robust.

Summary: Choosing a Classification Method

Method	Best For	Strengths	Weaknesses
LDA	Small samples, Gaussian data	Efficient, stable	Assumes normality
QDA	Different class shapes	Flexible boundaries	More parameters
Naive Bayes	High dimensions, text data	Fast, scales well	Wrong independence assumption
Logistic Regression	General purpose	Robust, probabilistic	Requires tuning
Perceptron	Historical interest	Simple	Unstable, no probabilities
SVM (Max Margin)	Separable data, small samples	Unique solution, sparse	Hard to interpret

For most practical applications, **logistic regression** is a great starting point. For high-dimensional text classification, **Naive Bayes** is surprisingly effective. For small samples with well-behaved data, **LDA** is worth trying.

ewpage

Kernel Methods

Kernel methods are a powerful family of techniques that enable us to work in high-dimensional (even infinite-dimensional!) feature spaces without explicitly computing the transformations. This chapter covers density estimation, classification using kernels, and radial basis functions.

The Big Picture

Many real-world patterns are non-linear. Kernel methods handle this by: 1. **Implicitly mapping** data to a higher-dimensional space where patterns become linear 2. **Using local information** — nearby points matter more than distant ones 3. **Avoiding the curse of dimensionality** through the "kernel trick"

Kernel Density Estimation

Before classifying data, we often need to estimate the underlying probability distribution. Kernel density estimation (KDE) is a non-parametric way to do this.

The Problem

Given a random sample $[x_1, x_2, ..., x_N]$ drawn from an unknown distribution $f_X(x)$, how do we estimate what f_X looks like?

A First Attempt: Histograms

The simplest approach is a histogram: divide the space into bins and count how many points fall in each bin. But histograms are bumpy and depend heavily on bin placement.

Parzen (Kernel) Density Estimation

A smoother approach: for any point x_0, estimate the density by looking at how many training points are nearby:

$$\hat{f}_X(x_0) = \frac{1}{N\lambda} \times \#\{x_i \in \text{neighborhood of } x_0\}$$

Where λ is the **bandwidth** — the width of the neighborhood.

Problem: This is still bumpy at the neighborhood boundaries.

Gaussian Kernels for Smooth Estimates

Instead of hard boundaries, use a **smooth kernel** that gives more weight to closer points:

$$\hat{f}_X(x_0) = \frac{1}{N}\sum_{i=1}^{N} K_\lambda(x_i, x_0)$$

Where the Gaussian kernel is:

$$K_\lambda(x_i, x_0) = \frac{1}{\lambda\sqrt{2\pi}} \exp\left(-\frac{\|x_i - x_0\|^2}{2\lambda^2}\right)$$

Intuition: We're placing a small "bump" (Gaussian) at each data point, then adding them all up. The result is a smooth density estimate.

The Effect of Bandwidth

- **Small λ**: Peaks around each data point, very bumpy (overfitting)
- **Large λ**: Very smooth, may miss important features (underfitting)
- **Just right λ**: Captures the true density shape

Choosing bandwidth is similar to choosing model complexity — cross-validation helps!

Mathematical View: Convolution

The kernel density estimate can be viewed as a **convolution** of the empirical distribution (spikes at each data point) with a Gaussian kernel:

$$\hat{f}_X(x_0) = \frac{1}{N}\sum_{i=1}^{N} \phi_\lambda(x_0 - x_i)$$

This "smears out" the point masses into a smooth function.

Kernel Density Classification

Now we can use density estimation for classification!

Bayes' Theorem for Classification

For class j:

$$P(G = j|X = x_0) \propto \hat{\pi}_j \times \hat{f}_j(x_0)$$

Where: - $\hat{\pi}_j$ = estimated prior probability (proportion of class j in training data) - $\hat{f}_j(x_0)$ = kernel density estimate for class j, evaluated at x_0

Algorithm: 1. Estimate the density separately for each class 2. Multiply by the class prior 3. Classify to the class with highest product

A Subtlety: Where Density Matters

Learning separate class densities everywhere can be misleading. Consider: - In dense regions (many training points), estimates are reliable - In sparse regions (few training points), estimates are noisy

Key insight: The density estimates only matter near the **decision boundary**. Far from the boundary, one class dominates anyway.

Naive Bayes Classifier

When Dimensions Are High

In high dimensions, kernel density estimation struggles — you need exponentially more data to fill the space (curse of dimensionality).

Naive Bayes makes a strong simplifying assumption: given the class, features are **conditionally independent**.

$$f_j(x) = \prod_{p=1}^{P} f_{jp}(x_p)$$

Breaking Down the Independence Assumption

Instead of estimating one P-dimensional density (hard), we estimate P one-dimensional densities (easy!).

For continuous features: Use univariate Gaussian or kernel density estimates for each feature.

For categorical features: Simply count proportions.

The Log-Odds Decomposition

$$\log \frac{P(G=k|X)}{P(G=l|X)} = \log \frac{\pi_k}{\pi_l} + \sum_{p=1}^{P} \log \frac{f_{kp}(x_p)}{f_{lp}(x_p)}$$

Each feature contributes additively to the log-odds — simple and interpretable!

Why "Naive" Works

The independence assumption is almost always wrong. Yet Naive Bayes often performs well because: 1. **We only need rankings**: For classification, we just need to rank classes correctly, not get exact probabilities 2. **Errors may cancel**: Overestimating some terms and underestimating others can balance out 3. **Robustness**: Fewer parameters means less overfitting

Best applications: Text classification (spam filtering, sentiment analysis), where features (words) are high-dimensional.

Radial Basis Functions (RBF)

Radial basis functions offer another approach: explicitly construct basis functions centered at various points, then fit a linear model in this new feature space.

The Idea of Basis Functions

Instead of modeling $f(x) = \beta^T x$ (linear in original features), use:

$$f(x) = \sum_{j=1}^{M} \beta_j h_j(x)$$

Where $h_j(x)$ are **basis functions** — transformations of the original features.

Why Higher Dimensions Help

Data that isn't linearly separable in the original space may become linearly separable in a higher-dimensional feature space.

Classic example: XOR problem. Points at (0,0) and (1,1) are class 1; points at (0,1) and (1,0) are class 2. No line separates them in 2D, but adding the feature $x_1 \cdot x_2$ makes separation easy!

Gaussian RBFs

RBF uses Gaussian kernels as basis functions:

$$f(x) = \sum_{j=1}^{M} \beta_j K_{\lambda_j}(\xi_j, x)$$

Where: - ξ_j = center of the j-th basis function (a point in feature space) - λ_j = width of the j-th kernel - β_j = coefficient (learned by regression)

More explicitly:

$$f(x) = \sum_{j=1}^{M} \beta_j \exp\left(-\frac{\|x - \xi_j\|^2}{2\lambda_j^2}\right)$$

Connection to Infinite Dimensions

The Gaussian kernel has a remarkable property: its Taylor series expansion involves polynomials of all degrees!

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

So using Gaussian RBFs is like using polynomial features of infinite degree — but without the computational explosion.

Fitting RBF Models

Full optimization: Learn β, ξ, and λ by minimizing squared error. - Non-linear in ξ and λ - Requires gradient descent or other iterative methods - Risk of local minima

Simpler approach: Fix ξ and λ, only learn β. - Choose centers using unsupervised methods (e.g., k-means on training data) - Use a constant bandwidth based on average distances - Then it's just linear regression!

Potential Pitfalls

If the basis function centers don't cover the input space well, there can be "holes" — regions where no kernel has significant weight, leading to poor predictions.

Gaussian Mixture Models

Mixture models extend RBFs to probabilistic density estimation.

The Model

$$f(x) = \sum_{j=1}^{M} \alpha_j \phi(x; \mu_j, \Sigma_j)$$

Where: - ϕ = Gaussian density function - α_j = mixing proportion (how much weight to give component j) - $\sum_j \alpha_j = 1$ - μ_j, Σ_j = mean and covariance of component j

Interpretation

The data is generated by: 1. Randomly selecting a component j with probability α_j 2. Drawing a point from the Gaussian $N(\mu_j, \Sigma_j)$

Each component represents a “cluster” or subpopulation in the data.

Connection to RBF

If we restrict covariances to be spherical ($\Sigma_j = \sigma^2 I$), mixture models reduce to radial basis expansions!

Fitting with Maximum Likelihood

Parameters are fit by maximizing the likelihood of the data. This is typically done using the **EM algorithm** (covered in Model Selection chapter).

Classification with Mixtures

For classification: 1. Fit a separate mixture model for each class 2. Apply Bayes’ theorem: $P(G = j|x) \propto \hat{\pi}_j \times \hat{f}_j(x)$

This is more flexible than single-Gaussian LDA — each class can have multiple modes.

Summary: When to Use What

Method	Best For	Pros	Cons
Kernel Density Estimation	Low dimensions, flexible shape	Non-parametric, intuitive	Curse of dimensionality
Naive Bayes	High dimensions, text/categorical	Fast, scales well	Wrong independence assumption
RBF Networks	Smooth non-linear functions	Flexible, local	Need to choose centers/widths

Method	Best For	Pros	Cons
Mixture Models	Multi-modal distributions	Probabilistic, interpretable	Need to choose number of components

Key Takeaways

1. **Kernels enable locality**: Nearby points matter more than distant ones
2. **Bandwidth/width controls bias-variance**: Too small = overfit, too large = underfit
3. **High dimensions are hard**: Naive Bayes and the kernel trick help
4. **Generative vs. Discriminative**: Kernel density classification is generative (models P(XIG)); logistic regression is discriminative (models P(GIX))

ewpage

Model Assessment and Selection

This chapter addresses one of the most important questions in machine learning: **How do we know if our model is any good?** Training error can be misleading, so we need principled ways to estimate how well our model will perform on new, unseen data.

The Big Picture

When building models, we face a fundamental tension: - **Simple models** may miss important patterns (underfitting) - **Complex models** may memorize noise (overfitting)

The goal is to find the sweet spot — a model complex enough to capture real patterns but simple enough to generalize to new data.

Understanding Generalization

What We Really Care About

We don't just want a model that fits our training data well. We want a model that predicts well on **new data** it hasn't seen before. This is called **generalization**.

Expected Test Error

The quantity we want to minimize:

$$\text{Err}_T = E[L(Y, \hat{f}(X))|T]$$

Where: - **T** = the training set used to build the model - **L** = a loss function measuring prediction error - The expectation is over new (X, Y) pairs

Common Loss Functions

For Regression: - **Squared Error**: $L(y, \hat{f}(x)) = (y - \hat{f}(x))^2$ - Most common; penalizes large errors heavily - **Absolute Error**: $L(y, \hat{f}(x)) = |y - \hat{f}(x)|$ - More robust to outliers

For Classification: - **0-1 Loss**: $L(y, \hat{G}(x)) = I(y \neq \hat{G}(x))$ - Simply counts misclassifications - **Deviance (Log-loss)**: $L(y, \hat{p}(x)) = -2 \log \hat{p}(x)$ - Penalizes confident wrong predictions heavily

---

The Bias-Variance Tradeoff

This is perhaps the most important concept in all of machine learning!

Decomposing Prediction Error

For squared error loss, we can decompose the expected prediction error at a point x_0:

$$\text{Err}(x_0) = E[(Y - \hat{f}(x_0))^2]$$

Assuming $Y = f(X) + \epsilon$ where $E[\epsilon] = 0$ and $\text{Var}(\epsilon) = \sigma_\epsilon^2$:

$$\text{Err}(x_0) = \underbrace{\sigma_\epsilon^2}_{\text{Irreducible}} + \underbrace{[E[\hat{f}(x_0)] - f(x_0)]^2}_{\text{Bias}^2} + \underbrace{E[(\hat{f}(x_0) - E[\hat{f}(x_0)])^2]}_{\text{Variance}}$$

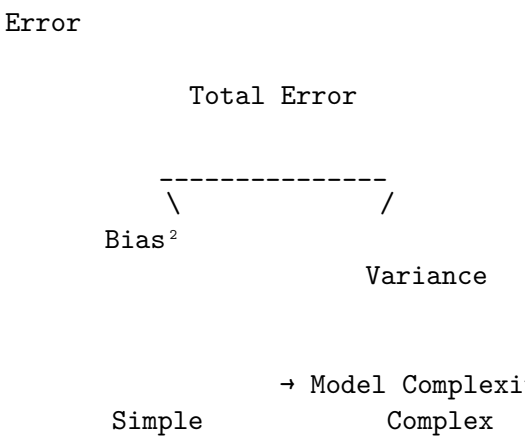
What Each Term Means

Irreducible Error ($\sigma^2\varepsilon$) - The inherent randomness in Y that no model can predict - Even with infinite data and a perfect model, you can't beat this - Sets the floor for prediction error

Bias² - How far off is our model *on average*? - Measures systematic error — does the model consistently over or underpredict? - Simple models tend to have high bias (they make strong assumptions that may be wrong)

Variance - How much does our model *fluctuate* across different training sets? - If you trained on different samples, how different would your predictions be? - Complex models tend to have high variance (they're sensitive to specific training data)

The Tradeoff Visualized



- **Simple models**: High bias, low variance (underfitting)
- **Complex models**: Low bias, high variance (overfitting)
- **Optimal**: Balance that minimizes total error

Practical Implications

1. **U-shaped test error curve**: As complexity increases, test error first decreases (reducing bias) then increases (increasing variance)
2. **Training error is misleading**: It always decreases with complexity — it can't detect overfitting!
3. **More data helps variance**: With more training data, variance decreases (we get more reliable estimates)
4. **Signal-to-noise ratio matters**: When noise is high, simpler models often win

---

Bias-Variance for Linear Models

For linear regression with p predictors:

Variance $\propto p$ - More parameters = more things to estimate = more variance

Bias² = Model Bias² + Estimation Bias² - **Model Bias**: Difference between the true function and the best linear approximation - **Estimation Bias**: Difference between what we estimate and the best linear approximation

For **OLS**: Estimation bias is zero (OLS gives unbiased estimates), but variance can be high.

For **Ridge Regression**: We *introduce* estimation bias deliberately to reduce variance. If the variance reduction outweighs the bias increase, we get lower overall error!

Why Training Error is Optimistic

The Fundamental Problem

Training error uses the same data to fit the model and evaluate it. The model is specifically tuned to do well on this data, so training error underestimates how the model will perform on new data.

Quantifying the Optimism

Define: - $\bar{\text{err}}$ = average training error - Err_{in} = expected error on training points (but with new Y values)

The **optimism** is:

$$\text{op} = \text{Err}_{in} - \bar{\text{err}} = \frac{2}{N}\sum_{i=1}^{N} \text{Cov}(y_i, \hat{y}_i)$$

Interpretation: Optimism measures how much each training label influences its own prediction. The more the model "memorizes" training labels, the more optimistic training error becomes.

What Affects Optimism?

- **More parameters** □ More optimism (more opportunity to fit training data specifically)
- **More training samples** □ Less optimism per point (each point has less influence)

Model Selection vs. Model Assessment

These are related but distinct tasks:

Model Selection - Goal: Choose the best model from a set of candidates - Question: Which model will generalize best? - Uses validation data

Model Assessment - Goal: Estimate how well the chosen model performs - Question: What's the expected error on new data? - Uses test data (must be kept separate from selection!)

The Standard Split

Set	Purpose	Typical Size
Training	Fit models	50-60%
Validation	Select best model	20-25%
Test	Estimate final performance	20-25%

Critical rule: Never use test data for model selection! This leads to optimistic error estimates.

Analytical Estimates of Prediction Error

When data is scarce, we'd rather not set aside a validation set. These methods estimate test error analytically.

Cp Statistic (Mallow's Cp)

$$C_p = \bar{\text{err}} + \frac{2p}{N}\hat{\sigma}_\epsilon^2$$

Where p = effective number of parameters.

Interpretation: Start with training error, add a penalty for complexity. The penalty estimates the optimism.

AIC (Akaike Information Criterion)

$$\text{AIC} = -\frac{2}{N}\ell + \frac{2p}{N}$$

Where ℓ = log-likelihood of the model.

How to use: Fit multiple models, compute AIC for each, choose the one with lowest AIC.

Properties: - Derived from information theory (minimizing KL divergence) - Tends to select slightly complex models - Works for any model with a likelihood (not just linear)

BIC (Bayesian Information Criterion)

$$\text{BIC} = -\frac{2}{N}\ell + \frac{\log N}{N} \times p$$

Difference from AIC: BIC penalizes complexity more heavily (especially for large N).

Properties: - Derived from Bayesian model comparison - **Consistent**: Will select the true model as N □ ∞ (if it's among the candidates) - Tends to select simpler models than AIC

AIC vs BIC: When to Use Which?

Criterion	Penalty	Tends to Select	Best When
AIC	2p/N	Larger models	Prediction is goal
BIC	(log N)p/N	Smaller models	Want to find "true" model

Effective Number of Parameters

For complex models, "number of parameters" isn't straightforward.

General definition: For any smoother $\hat{y} = Sy$, the effective degrees of freedom is:

$$\text{df} = \text{trace}(S)$$

For OLS: df = p (the number of predictors + intercept)

For Ridge Regression:

$$\text{df}(\lambda) = \sum_{j=1}^{p} \frac{d_j^2}{d_j^2 + \lambda}$$

Where d_j are the eigenvalues of $X^T X$. As λ increases, effective df decreases.

VC Dimension

For non-linear models, counting parameters is inadequate. The **VC (Vapnik-Chervonenkis) dimension** provides a general measure of model complexity.

The Concept of Shattering

A set of points is **shattered** by a class of functions if, for every possible labeling of the points, some function in the class can achieve that labeling perfectly.

Example: 3 points in 2D - Linear classifiers can separate any labeling of 3 non-collinear points - So linear classifiers in 2D can shatter 3 points - But they cannot shatter 4 points (XOR configuration is impossible)

VC Dimension Defined

The VC dimension of a class of functions is the **largest number of points** that can be shattered.

Examples: - Linear classifiers in d dimensions: VC = d + 1 - Polynomials of degree k: VC = k + 1 - Neural networks: depends on architecture (roughly proportional to number of parameters)

Structural Risk Minimization

The VC dimension gives bounds on generalization error:

$$\text{Test Error} \leq \text{Training Error} + \sqrt{\frac{h(\log(2N/h)+1) - \log(\eta/4)}{N}}$$

Where h = VC dimension, N = sample size, η = confidence level.

Implication: Models with smaller VC dimension have tighter bounds — they're more likely to generalize well.

Cross-Validation

When you can't afford separate validation data (or want to use all data efficiently), **cross-validation** provides a powerful alternative.

K-Fold Cross-Validation

1. Randomly divide data into K equal parts (folds)
2. For k = 1 to K:
   - Train on all folds except k
   - Test on fold k
   - Record error
3. Average the K test errors

$$\text{CV}_K = \frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{f}^{-\kappa(i)}(x_i))$$

Where $\hat{f}^{-\kappa(i)}$ means the model trained without observation i's fold.

Common Choices of K

K = N (Leave-One-Out CV) - Train on N-1 points, test on 1 point, repeat N times - Nearly unbiased (training set almost full size) - High variance (test sets overlap heavily) - Computationally expensive (unless there's a shortcut formula)

K = 5 or 10 (The Sweet Spot) - Good balance of bias and variance - Computationally reasonable - Empirically shown to work well - Most commonly recommended

K = 2 (Repeated Random Splits) - High variance - Not commonly used except with many repetitions

Bias-Variance in Cross-Validation

K	Training Size	Bias	Variance
Small (e.g., 2)	~N/2	High (small training sets)	Lower
Large (e.g., N)	N-1	Low (big training sets)	High (overlapping test sets)
5-10	~80-90% of N	Moderate	Moderate

Bootstrap Methods

The bootstrap is a general technique for estimating uncertainty by resampling.

The Bootstrap Idea

1. Draw B samples of size N **with replacement** from training data
2. Fit a model to each bootstrap sample
3. Use the variation across fits to estimate uncertainty

Key Property

Each bootstrap sample contains about 63.2% unique observations:

$$P(\text{observation } i \text{ is in bootstrap sample}) = 1 - \left(1 - \frac{1}{N}\right)^N \approx 1 - e^{-1} \approx 0.632$$

Estimating Prediction Error

Naive approach: Average error on training points across bootstrap samples. - **Problem**: Observations often appear in both training and test — leakage!

Out-of-Bag (OOB) error: For each observation, only use predictions from models where that observation wasn't in the training sample.

$$\widehat{\text{Err}}^{\text{OOB}} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{|C^{-i}|}\sum_{b\in C^{-i}} L(y_i, \hat{f}^{*b}(x_i))$$

Where C^{-i} = bootstrap samples not containing observation i.

Properties of OOB error: - No leakage — honest estimate - Similar to leave-one-out CV - Free when using bagging/random forests

The .632 Bootstrap Estimator

OOB error can be slightly pessimistic (training sets are only ~63% of full data). A correction:

$$\widehat{\text{Err}}^{.632} = 0.368 \times \bar{\text{err}} + 0.632 \times \widehat{\text{Err}}^{\text{OOB}}$$

This averages training error (too optimistic) with OOB error (slightly pessimistic).

Summary: Choosing an Estimation Method

Method	When to Use	Pros	Cons
Train/Val/Test Split	Lots of data	Simple, unbiased	Wastes data
AIC/BIC	Comparing similar models, need speed	Fast, no retraining	Approximate, limited scope
Cross-Validation	Limited data, want reliability	Uses all data for training AND testing	Computationally expensive
Bootstrap	Need uncertainty estimates	Versatile, works for any statistic	Can be biased, requires care

Practical Recommendations

1. **If you have lots of data**: Use a held-out test set for final assessment
2. **For model selection with limited data**: Use 5-fold or 10-fold CV
3. **For very small samples**: Use leave-one-out CV or bootstrap
4. **Quick comparisons**: AIC/BIC are fast approximations
5. **Always**: Keep a final test set untouched until the very end!

Common Pitfalls

- **Using test data for selection**: Invalidates your error estimate
- **Ignoring the bias-variance tradeoff**: Don't just minimize training error!
- **Forgetting about randomness**: Always think about how results would vary with different data

ewpage

Model Inference and Averaging

This chapter covers the statistical foundations of model fitting and methods for combining multiple models. We'll explore Maximum Likelihood Estimation, Bayesian methods, the EM algorithm, and Markov Chain Monte Carlo (MCMC) — tools that underpin much of modern machine learning.

The Big Picture

So far we've focused on finding the "best" model. But we haven't asked: - **How confident are we** in our parameter estimates? - **What if the data came from a mixture** of underlying processes? - **Can we combine multiple models** for better predictions?

This chapter provides the statistical machinery to answer these questions.

Maximum Likelihood Estimation (MLE)

MLE is the most widely used approach for fitting statistical models. The intuition is simple: choose parameters that make the observed data most likely.

The Setup

We have: - Data: $Z = \{z_1, z_2, ..., z_N\}$ - Parametric model: $z_i \sim g_\theta(z)$ - Unknown parameters: θ (could be μ, σ^2, etc.)

The Likelihood Function

The likelihood is the probability of observing our data, viewed as a function of the parameters:

$$L(\theta; Z) = \prod_{i=1}^{N} g_\theta(z_i)$$

Key insight: The likelihood treats the data as fixed and varies the parameters. It answers: "For different values of θ, how probable was it to see exactly this data?"

Log-Likelihood

Products are numerically unstable, so we usually work with the log-likelihood:

$$\ell(\theta; Z) = \sum_{i=1}^{N} \log g_\theta(z_i)$$

Taking logs turns products into sums — much easier to optimize!

Finding the MLE

The maximum likelihood estimate $\hat{\theta}$ maximizes $L(\theta)$ (equivalently, $\ell(\theta)$):

$$\hat{\theta} = \arg\max_\theta \ell(\theta; Z)$$

For many distributions (Normal, Binomial, etc.), we can solve this analytically by: 1. Taking the derivative (the **score function**): $S(\theta) = \frac{\partial \ell}{\partial \theta}$ 2. Setting it to zero: $S(\hat{\theta}) = 0$ 3. Solving for $\hat{\theta}$

The Information Matrix

How curved is the log-likelihood around the maximum? This tells us how “sharp” or “flat” the peak is — sharper = more confident in our estimate.

The **Information Matrix** captures this curvature:

$$I(\theta) = -E\left[\frac{\partial^2 \ell}{\partial \theta^2}\right]$$

Fisher Information is this evaluated at the MLE: $i(\theta) = I(\theta)|_{\hat{\theta}}$

Sampling Distribution of the MLE

Under regularity conditions, as N □ ∞:

$$\hat{\theta} \sim N(\theta, I(\theta)^{-1})$$

What this means: - The MLE is asymptotically unbiased (centered on truth) - Its variance is the inverse of the information matrix - We can construct confidence intervals and hypothesis tests!

Example: Linear Regression

For linear regression with Gaussian errors, OLS gives the MLE:

$$\text{Var}(\hat{\beta}) = \sigma^2 (X^T X)^{-1}$$

$$\text{Var}(\hat{y}_i) = \sigma^2 X_i^T (X^T X)^{-1} X_i$$

For non-Gaussian errors, OLS is still unbiased but may not be the most efficient estimator.

Bootstrap

The bootstrap is a powerful resampling technique that provides uncertainty estimates when theoretical formulas don't exist (or are too complex).

The Core Idea

Pretend your training data IS the population. Resample from it with replacement to simulate "new datasets." The variation across these resampled datasets estimates uncertainty.

Non-Parametric Bootstrap

1. Sample N observations **with replacement** from your data
2. Fit your model to this bootstrap sample
3. Repeat B times (typically B = 100-1000)
4. The distribution of estimates approximates the sampling distribution

Common approaches: - **Case resampling**: Sample entire (X, Y) pairs with replacement - **Residual resampling**: Fit model, resample residuals, add to fitted values

Parametric Bootstrap

1. Fit model to original data
2. Simulate new data from the fitted model (add Gaussian noise to predictions)
3. Refit model to simulated data
4. Repeat and analyze distribution

This assumes you know the correct error distribution (often Gaussian).

Why Bootstrap Works

Under certain conditions, the bootstrap distribution of $\widehat{\theta}^* - \widehat{\theta}$ (difference between bootstrap and original estimate) approximates the true sampling distribution of $\widehat{\theta} - \theta$.

Bootstrap Confidence Intervals

Percentile Method: Use the 2.5th and 97.5th percentiles of bootstrap estimates as a 95% CI.

BCa Method (Bias-Corrected and Accelerated): Adjusts for bias and skewness — more accurate but more complex.

Connection to Bayesian Inference

There's a remarkable connection: under certain priors, the bootstrap distribution approximates the Bayesian posterior distribution!

Bagging: Bootstrap for Prediction

Bagging (Bootstrap Aggregating) averages predictions across bootstrap samples:

1. Generate B bootstrap samples
2. Fit a model to each
3. Average predictions (regression) or vote (classification)

Why it helps: Reduces variance, especially for unstable models like decision trees.

Bayesian Methods

Bayesian inference takes a fundamentally different view: parameters are random variables with their own probability distributions.

Prior, Likelihood, Posterior

Prior $P(\theta)$ - What we believe about parameters BEFORE seeing data - Encodes prior knowledge or assumptions

Likelihood $P(Z|\theta)$ - Probability of the data given parameters - Same as in MLE

Posterior $P(\theta|Z)$ - Updated beliefs AFTER seeing data - This is what we want!

Bayes' Theorem:

$$P(\theta|Z) \propto P(Z|\theta) \times P(\theta)$$

"Posterior is proportional to Likelihood times Prior"

Types of Priors

Informative Priors: Strong beliefs based on domain knowledge - Example: "The coefficient is probably between 0 and 1"

Non-informative Priors: Minimal assumptions - Example: Uniform distribution over all possible values - Let the data speak!

Conjugate Priors: Mathematical convenience — the posterior has the same form as the prior - Example: Gaussian prior + Gaussian likelihood □ Gaussian posterior

The Posterior Distribution

Unlike MLE (which gives a single point estimate), Bayesian inference gives a **full distribution** over parameters. This lets us: - Quantify uncertainty directly - Make probabilistic statements ("there's a 95% probability that β is between 0.3 and 0.7") - Incorporate prior knowledge naturally

Predictive Distribution

To predict a new observation, we don't just plug in a single parameter value. We integrate over all possible values:

$$P(z_{\text{new}}|Z) = \int P(z_{\text{new}}|\theta)P(\theta|Z)d\theta$$

This **accounts for parameter uncertainty** — predictions are more honest about what we don't know.

MAP Estimation

Maximum A Posteriori (MAP) is a compromise: find the single most probable parameter value under the posterior.

$$\hat{\theta}^{MAP} = \arg\max_{\theta} P(\theta|Z) = \arg\max_{\theta}[P(Z|\theta) \cdot P(\theta)]$$

Or equivalently:

$$\hat{\theta}^{MAP} = \arg\max_{\theta}[\log P(Z|\theta) + \log P(\theta)]$$

Key insight: MAP = MLE + regularization penalty from the prior!

- **Gaussian prior** □ L2 penalty □ Ridge Regression
- **Laplace prior** □ L1 penalty □ Lasso Regression

Hierarchical Bayesian Models

Sometimes we have grouped data (e.g., students within schools). **Hierarchical models** place priors on hyperparameters, allowing "borrowing strength" across groups.

Example: Estimating school-level effects - Each school has its own mean μ_k - But the μ_k's come from a common distribution N(μ, τ²) - We estimate μ and τ² from all schools together

This naturally handles the bias-variance tradeoff across groups!

The EM Algorithm

The **Expectation-Maximization (EM)** algorithm is an elegant solution for maximum likelihood when data is "incomplete" — either literally missing or involving latent (hidden) variables.

Motivating Example: Gaussian Mixture Models

Suppose your data comes from a mixture of two Gaussians: - Component 1: $N(\mu_1, \sigma_1^2)$ with probability π - Component 2: $N(\mu_2, \sigma_2^2)$ with probability $1 - \pi$

The density is:

$$g(y) = (1 - \pi)\phi_1(y) + \pi\phi_2(y)$$

The Problem with Direct MLE

The log-likelihood is:

$$\ell(\theta) = \sum_{i=1}^{N} \log[(1 - \pi)\phi_1(y_i) + \pi\phi_2(y_i)]$$

The sum is INSIDE the log — no nice closed-form solution!

The Latent Variable Perspective

Imagine we knew which component each observation came from (a latent indicator $\Delta_i \in \{0, 1\}$). Then MLE would be easy!

The EM insight: We don't know Δ_i, but we can compute its expected value given current parameter estimates.

The Algorithm

1. Initialize: Start with guesses for all parameters (e.g., sample means and variances)

2. E-Step (Expectation): Compute "soft" assignments — the probability each point belongs to each component:

$$\gamma_i = P(\Delta_i = 1|y_i, \theta) = \frac{\hat{\pi}\phi_2(y_i)}{(1 - \hat{\pi})\phi_1(y_i) + \hat{\pi}\phi_2(y_i)}$$

This is called the **responsibility** — how responsible is component 2 for observation i?

3. M-Step (Maximization): Update parameters using weighted averages:

$$\hat{\mu}_1 = \frac{\sum(1 - \gamma_i)y_i}{\sum(1 - \gamma_i)}$$

$$\hat{\mu}_2 = \frac{\sum \gamma_i y_i}{\sum \gamma_i}$$

$$\hat{\pi} = \frac{\sum \gamma_i}{N}$$

4. Repeat until convergence (parameters stop changing).

Key Properties of EM

1. **Monotonic**: Each iteration increases the likelihood (never goes down)
2. **Converges**: Always reaches a fixed point
3. **Local optima**: May not find the global maximum — try multiple initializations!
4. **Slow near convergence**: Can take many iterations to converge precisely

Applications of EM

- **Mixture models**: Clustering with soft assignments
- **Missing data**: Impute missing values, then estimate
- **Hidden Markov Models**: Speech recognition, genomics
- **Factor analysis**: Latent variable models

Markov Chain Monte Carlo (MCMC)

When posteriors are too complex for closed-form solutions, MCMC provides a way to **sample** from them numerically.

The Challenge

In Bayesian inference, we need:

$$P(\theta|Z) = \frac{P(Z|\theta)P(\theta)}{P(Z)}$$

But the denominator $P(Z) = \int P(Z|\theta)P(\theta)d\theta$ is often intractable!

The MCMC Idea

Instead of computing the posterior exactly, generate **samples** from it. With enough samples, we can estimate anything we want (means, quantiles, etc.).

Gibbs Sampling

When you have multiple parameters and can sample from **conditional distributions** (one parameter at a time, given all others):

1. Initialize all parameters: $\theta^{(0)} = (\theta_1^{(0)}, \theta_2^{(0)}, ..., \theta_K^{(0)})$
2. For each iteration t:
   - Sample $\theta_1^{(t+1)} \sim P(\theta_1|\theta_2^{(t)}, \theta_3^{(t)}, ..., \theta_K^{(t)}, Z)$
   - Sample $\theta_2^{(t+1)} \sim P(\theta_2|\theta_1^{(t+1)}, \theta_3^{(t)}, ..., \theta_K^{(t)}, Z)$
   - …and so on for all parameters
3. After a burn-in period, keep the samples

Key property: The sequence of samples forms a Markov chain whose stationary distribution is the true joint posterior!

Metropolis-Hastings Algorithm

A more general MCMC approach:

1. **Propose** a new value: $\theta^* \sim q(\theta^*|\theta^{(t)})$
2. **Compute** acceptance ratio:

$$r = \min\left(1, \frac{P(\theta^*|Z) \cdot q(\theta^{(t)}|\theta^*)}{P(\theta^{(t)}|Z) \cdot q(\theta^*|\theta^{(t)})}\right)$$

3. **Accept** θ^* with probability r; otherwise keep $\theta^{(t)}$

Special cases: - Random walk proposals: $q(\theta^*|\theta) = N(\theta, \sigma^2)$ - Independent proposals: $q(\theta^*|\theta) = g(\theta^*)$

Practical Considerations

Burn-in: Discard early samples (they depend on initialization)

Thinning: Keep every k-th sample to reduce autocorrelation

Convergence diagnostics: - Trace plots: Should look like “noise” around a stable value - Gelman-Rubin statistic: Compare multiple chains

EM vs. MCMC

Aspect	EM	MCMC
Output	Point estimate (mode)	Samples from full posterior
Speed	Usually faster	Can be slow
Uncertainty	Limited	Full posterior available
Local optima	Can get stuck	Explores full space

Summary: Choosing an Inference Method

Method	When to Use	What You Get
MLE	Large samples, well-specified model	Point estimate + asymptotic inference
Bootstrap	Unknown distribution, complex statistics	Empirical confidence intervals
Bayesian + MCMC	Prior knowledge, need full uncertainty	Full posterior distribution
EM	Latent variables, mixture models	MLE for incomplete data
MAP	Want regularization with Bayesian interpretation	Regularized point estimate

Key Takeaways

1. **MLE is the workhorse**: Simple, interpretable, asymptotically optimal
2. **Bootstrap is versatile**: Works when theory fails
3. **Bayesian methods quantify uncertainty**: But require prior choices
4. **EM handles hidden structure**: Essential for clustering and missing data
5. **MCMC explores complex posteriors**: Powerful but computationally intensive

ewpage

Additive Models, Trees, and Related Methods

This chapter introduces flexible methods that can capture non-linear relationships while remaining interpretable. We start with generalized additive models, then dive deep into decision trees — one of the most intuitive and widely-used machine learning methods.

The Big Picture

Linear models are simple and interpretable but can miss important non-linear patterns. At the other extreme, highly flexible methods (like neural networks) can fit anything but are hard to interpret.

Additive models and trees offer a middle ground: they capture non-linear relationships while remaining interpretable.

Generalized Additive Models (GAMs)

The Limitation of Linear Models

In linear regression, we model:

$$E[Y|X] = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p$$

Each predictor has a simple linear effect. But what if the relationship is curved? What if income affects health differently for low vs. high earners?

The GAM Solution

Replace linear terms with **flexible smooth functions**:

$$E[Y|X] = \alpha + f_1(X_1) + f_2(X_2) + ... + f_p(X_p)$$

Where each f_j is learned from data — typically a smooth curve like a **spline**.

More Generally: Link Functions

For non-normal responses (binary, count data), we use a link function:

$$g[E[Y|X]] = \alpha + f_1(X_1) + f_2(X_2) + ... + f_p(X_p)$$

Common link functions: - **Identity** (linear regression): $g(\mu) = \mu$ - **Logit** (logistic regression): $g(\mu) = \log(\mu/(1-\mu))$ - **Log** (Poisson regression): $g(\mu) = \log(\mu)$

Interpretability

Key advantage: Each f_j shows exactly how predictor X_j affects the response. You can plot $f_j(X_j)$ and see the relationship!

- Is it linear? Curved? Has a threshold?
- The shape is learned from data, not assumed

Fitting GAMs

GAMs are fit by minimizing **Penalized Residual Sum of Squares (PRSS)**, which balances fitting the data with keeping functions smooth:

$$\text{PRSS} = \sum_{i=1}^{N} \left(y_i - \alpha - \sum_j f_j(x_{ij}) \right)^2 + \sum_j \lambda_j \int f_j''(t)^2 dt$$

The penalty term punishes "wiggly" functions (large second derivatives).

Decision Trees

Decision trees are perhaps the most intuitive machine learning method. They partition the feature space into regions and fit simple models (often just constants) in each region.

The Core Idea

Think of playing "20 Questions" with your data: - "Is income > $50K?" □ split data into two groups - "Is age > 40?" □ further split - Keep splitting until groups are "pure" (mostly one class/similar values)

The result is a tree structure that's easy to understand and explain.

Regression Trees

For continuous outcomes, we: 1. **Partition** the feature space into rectangles $R_1, R_2, ..., R_M$ 2. **Fit a constant** in each region: $c_m =$ average of y_i in R_m

The model is:

$$f(X) = \sum_{m=1}^{M} c_m \cdot I\{X \in R_m\}$$

Where $I\{\cdot\}$ is the indicator function (1 if true, 0 otherwise).

How to Find the Best Splits

We use a **greedy algorithm** — at each node, find the split that most reduces error.

For a split on variable X_j at value s: - Left region: $R_1 = \{X|X_j \leq s\}$ - Right region: $R_2 = \{X|X_j > s\}$

Choose j and s to minimize:

$$\min_{j,s} \left[\min_{c_1} \sum_{X_i \in R_1} (y_i - c_1)^2 + \min_{c_2} \sum_{X_i \in R_2} (y_i - c_2)^2 \right]$$

The inner minimizations are easy — just take averages in each region!

Classification Trees

For categorical outcomes, each leaf predicts the **most common class** in that region:

$$\hat{G}_m = \text{majority class in } R_m$$

The proportion of class k in node m is:

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{i \in R_m} I\{y_i = k\}$$

Splitting Criteria for Classification

We need to measure how "impure" a node is. Several options:

Misclassification Error: $1 - \max_k \hat{p}_{mk}$ - Simple but not differentiable — hard to optimize

Gini Index: $\sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$ - Measures probability of misclassifying a randomly chosen element - Equals variance of Bernoulli distribution when K=2 - Most commonly used

Cross-Entropy (Deviance): $-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$ - Information-theoretic measure of impurity - Similar to Gini in practice

Note: Gini and entropy are more sensitive to node purity changes than misclassification error, making them better for tree growing.

Handling Categorical Predictors

If variable X has L categories, there are $2^{L-1} - 1$ possible binary splits. This seems exponential, but for classification with 2 classes:

Trick: Order categories by proportion of class 1, then treat as ordered. This reduces to checking L-1 splits!

Handling Missing Values

Two common approaches:

1. Missing as a Category: Create a new category for missing values.

2. Surrogate Splits: Find alternative splits that mimic the primary split. At prediction time, if the primary split variable is missing, use the surrogate.

The surrogate approach leverages correlations between predictors to minimize information loss.

Pruning: Controlling Tree Size

The Problem

Trees that grow too large: - Overfit the training data - Have high variance - Are harder to interpret

Option 1: Stop Early

Split only if improvement exceeds a threshold.

Problem: A bad split now might enable great splits later! This is short-sighted.

Option 2: Grow and Prune (Better!)

1. **Grow** a large tree (until leaves have few observations)
2. **Prune** back to find the best subtree

Cost-Complexity Pruning

Define a cost that balances fit and complexity:

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T|$$

Where: - $|T|$ = number of terminal nodes (leaves) - Q_m = impurity measure for node m (e.g., RSS/N_m for regression) - α = complexity penalty parameter

Small α: Prefer large trees (focus on fit) **Large α**: Prefer small trees (focus on simplicity)

Algorithm: 1. For each α, find the subtree that minimizes $C_\alpha(T)$ 2. Use cross-validation to select the best α

Evaluating Classification Performance

The Confusion Matrix

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Key Metrics

Sensitivity (Recall, True Positive Rate):

$$\text{Sensitivity} = \frac{TP}{TP + FN}$$

"Of all actual positives, what fraction did we catch?"

Specificity (True Negative Rate):

$$\text{Specificity} = \frac{TN}{TN + FP}$$

"Of all actual negatives, what fraction did we correctly identify?"

Precision:

$$\text{Precision} = \frac{TP}{TP + FP}$$

"Of all predicted positives, what fraction are correct?"

The ROC Curve

The **Receiver Operating Characteristic (ROC)** curve shows the tradeoff between sensitivity and specificity as you vary the classification threshold.

- **X-axis**: 1 - Specificity (False Positive Rate)
- **Y-axis**: Sensitivity (True Positive Rate)

AUC (Area Under the ROC Curve): - AUC = 1: Perfect classifier - AUC = 0.5: Random guessing - AUC > 0.7: Generally acceptable

Interpretation: AUC is the probability that a randomly chosen positive example ranks higher than a randomly chosen negative example.

MARS: Multivariate Adaptive Regression Splines

MARS extends tree ideas to regression with smoother, continuous functions.

The Idea

Instead of piecewise constant regions, use **piecewise linear basis functions**:

$$(x - t)_+ = \max(0, x - t)$$
$$(t - x)_+ = \max(0, t - x)$$

These are "hockey stick" functions that are zero until a knot t, then linear.

MARS Model

$$f(X) = \beta_0 + \sum_{m=1}^{M} \beta_m h_m(X)$$

Where each h_m is a product of basis functions (allowing interactions).

Connection to Trees

MARS is like a regression tree with smoother predictions at boundaries. Trees have sharp jumps; MARS transitions smoothly.

PRIM: Patient Rule Induction Method

PRIM takes a different approach: find regions (boxes) with unusually high (or low) response values.

The Algorithm

1. Start with a box containing all data
2. **Peeling**: Shrink the box by removing a thin slice from one face, choosing the slice that maximizes mean response
3. **Pasting**: Try expanding the box if it improves the mean
4. Repeat to find multiple boxes

Use Case

Useful for finding "hot spots" — regions where the response is particularly high. Applications include fraud detection, medical diagnosis, quality control.

Mixture of Experts

This is a probabilistic generalization of decision trees.

The Idea

Instead of hard splits (left or right), use **soft probabilistic splits**: - Each observation has some probability of going to each child node - “Gating networks” at internal nodes determine these probabilities - “Expert” models at leaves make predictions - Final prediction is a weighted average

Structure

Gating Networks (internal nodes): - Soft decision functions - Output probabilities for each branch

Experts (terminal nodes): - Fit local models (often linear regression) - Each expert specializes in a region

Fitting

Use the **EM algorithm**: - E-step: Compute responsibilities (how much each expert contributes to each observation) - M-step: Update expert parameters and gating parameters

Advantages

- Smoother predictions than hard trees
- Naturally provides uncertainty estimates
- Can capture complex interactions

Summary: Choosing a Method

Method	Best For	Interpretability	Flexibility
GAM	Understanding non-linear effects	High (plot each effect)	Medium
Decision Tree	Simple rules, categorical outcomes	Very High	Medium
MARS	Regression with interactions	Medium	Medium-High
PRIM	Finding high-response regions	High	Low
Mixture of Experts	Complex boundaries with uncertainty	Low	High

Key Takeaways

1. **Trees are intuitive**: Easy to explain and visualize
2. **Pruning is essential**: Unpruned trees overfit badly
3. **GAMs maintain interpretability**: Each predictor's effect is visible
4. **Splitting criteria matter**: Gini/entropy better than misclassification for tree growing
5. **These methods form building blocks**: Trees are the foundation for Random Forests and Boosting!

ewpage

Boosting and Additive Trees

Boosting is one of the most powerful and widely-used machine learning techniques. The core idea is simple yet profound: combine many "weak" learners (models that are only slightly better than random guessing) into one strong learner.

The Big Picture

Imagine you're trying to predict whether a customer will churn. A single simple rule ("customers with low usage churn") might be 55% accurate. Not impressive! But what if you combine 100 such simple rules, each focusing on different aspects? The combination can be remarkably accurate.

Boosting does exactly this — it sequentially builds simple models, each one focusing on the mistakes of the previous ones.

AdaBoost: The Original Boosting Algorithm

AdaBoost (Adaptive Boosting) was one of the first successful boosting algorithms, and it remains one of the most elegant.

The Setup

- **Goal**: Binary classification with $Y \in \{-1, +1\}$
- **Weak learners**: Simple classifiers $G_m(x)$ (e.g., decision stumps — trees with just one split)
- **Final classifier**: Weighted vote of weak learners

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

The Algorithm Step by Step

1. Initialize: Give all observations equal weight

$$w_i = \frac{1}{N} \quad \text{for } i = 1, 2, ..., N$$

2. For m = 1 to M rounds:

a) **Fit** a weak classifier $G_m(x)$ using weights w_i

b) **Compute weighted error**:

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i \cdot I\{y_i \neq G_m(x_i)\}}{\sum_{i=1}^{N} w_i}$$

c) **Compute classifier weight**:

$$\alpha_m = \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right)$$

d) **Update observation weights**:

$$w_i \leftarrow w_i \cdot \exp\left(\alpha_m \cdot I\{y_i \neq G_m(x_i)\}\right)$$

e) **Normalize** weights so they sum to 1

3. Output: $G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$

Understanding the Weight Updates

The key insight is in step 2d: - **Correctly classified points**: Weight stays the same - **Misclassified points**: Weight increases by factor $\exp(\alpha_m)$

Since α_m is larger when err_m is smaller (i.e., when the classifier is better), good classifiers upweight their mistakes MORE. This forces the next classifier to focus on the hard cases!

Understanding Classifier Weights

The weight $\alpha_m = \log \frac{1-\text{err}_m}{\text{err}_m}$ has nice properties: - $\text{err}_m = 0.5$ (random guessing): $\alpha_m = 0$ (no contribution) - $\text{err}_m < 0.5$ (better than random): $\alpha_m > 0$ (positive contribution) - err_m near 0 (very accurate): α_m is large (strong contribution)

Why AdaBoost Works: Key Properties

1. **Adaptive**: Each round focuses on previously misclassified points
2. **Resistant to overfitting**: Empirically works well even with many rounds
3. **Theoretical guarantee**: Training error decreases exponentially with rounds:

$$\text{Training error} \leq \prod_{m=1}^{M} \sqrt{4 \cdot \text{err}_m \cdot (1 - \text{err}_m)}$$

Forward Stagewise Additive Modeling

AdaBoost can be understood as a special case of a general framework called **forward stagewise additive modeling**.

The General Framework

We want to build a model as a sum of basis functions:

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m)$$

Where: - $b(x; \gamma)$ is a basis function parameterized by γ - β_m is the coefficient for the m-th term

The Stagewise Algorithm

Instead of optimizing all parameters at once (hard!), we build the model **greedily**:

For m = 1 to M: 1. Fix previous terms $f_{m-1}(x)$ 2. Find new β_m, γ_m to minimize:

$$\sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta_m b(x_i; \gamma_m))$$

3. Update: $f_m(x) = f_{m-1}(x) + \beta_m b(x_i; \gamma_m)$

L2 Loss: Fitting Residuals

For squared error loss $L(y, f) = (y - f)^2$:

$$\min_{\beta, \gamma} \sum_{i=1}^{N} (y_i - f_{m-1}(x_i) - \beta b(x_i; \gamma))^2$$

Let $r_{im} = y_i - f_{m-1}(x_i)$ be the **residual** from previous rounds. Then:

$$\min_{\beta, \gamma} \sum_{i=1}^{N} (r_{im} - \beta b(x_i; \gamma))^2$$

Insight: With squared error, each round fits the **residuals** from the previous round!

Robust Loss Functions

Squared error is sensitive to outliers. Alternatives:

Huber Loss:

$$L(y, f) = \begin{cases} \frac{1}{2}(y - f)^2 & \text{if } |y - f| \leq \delta \\ \delta |y - f| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

- Quadratic for small errors (smooth optimization)
- Linear for large errors (robust to outliers)

Exponential Loss: Back to AdaBoost

For exponential loss $L(y, f) = \exp(-y \cdot f)$:

$$\min_{\beta, G} \sum_{i=1}^{N} \exp(-y_i(f_{m-1}(x_i) + \beta G(x_i)))$$

Let $w_i^{(m)} = \exp(-y_i f_{m-1}(x_i))$. Then:

$$\min_{\beta, G} \sum_{i=1}^{N} w_i^{(m)} \exp(-y_i \beta G(x_i))$$

Solving this optimization gives exactly the AdaBoost update rules!

Why Exponential Loss?

The population minimizer of exponential loss is:

$$f^*(x) = \frac{1}{2} \log \frac{P(Y = 1|X = x)}{P(Y = -1|X = x)}$$

This is half the log-odds! So $\text{sign}(f(x))$ gives the Bayes optimal classifier.

Gradient Boosting

Gradient boosting generalizes boosting to any differentiable loss function by viewing the problem as **gradient descent in function space**.

The Key Insight

Think of the fitted values $\mathbf{f} = [f(x_1), f(x_2), ..., f(x_N)]$ as parameters we're optimizing.

To minimize $L(\mathbf{f}) = \sum_{i=1}^{N} L(y_i, f(x_i))$: - Compute the gradient: $g_i = \left.\frac{\partial L(y_i, f)}{\partial f}\right|_{f=f_{m-1}(x_i)}$ - Update: $\mathbf{f}_{new} = \mathbf{f}_{old} - \rho \cdot \mathbf{g}$

The Problem

We can compute optimal f values for training points, but we need to generalize to new points!

The Solution: Fit a Model to the Gradient

Instead of using the gradient directly, we **fit a base learner to approximate the negative gradient**:

1. Compute negative gradient (pseudo-residuals):

$$r_{im} = -\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\bigg|_{f=f_{m-1}}$$

2. Fit a base learner $h_m(x)$ to the pseudo-residuals
3. Find optimal step size: $\rho_m = \arg\min_\rho \sum_i L(y_i, f_{m-1}(x_i) + \rho h_m(x_i))$
4. Update: $f_m(x) = f_{m-1}(x) + \rho_m h_m(x)$

Gradients for Common Loss Functions

Loss	Formula	Negative Gradient (Pseudo-residual)
L2 (Squared)	$(y - f)^2$	$y_i - f(x_i)$ (ordinary residual)
L1 (Absolute)	$\|y - f\|$	$\text{sign}(y_i - f(x_i))$
Deviance (Classification)	$-y \log p - (1-y) \log(1-p)$	$y_i - p(x_i)$
Huber	Piecewise	Trimmed residual

Gradient Boosting with Trees

The most common base learner is a **regression tree**. This combination is called **Gradient Boosted Trees (GBT)** or **Gradient Boosted Decision Trees (GBDT)**.

Algorithm: 1. Initialize $f_0(x) = \arg\min_\gamma \sum_i L(y_i, \gamma)$ 2. For m = 1 to M: - Compute pseudo-residuals: $r_{im} = -\frac{\partial L}{\partial f(x_i)}|_{f_{m-1}}$ - Fit tree h_m to pseudo-residuals - For each leaf j, compute optimal value: $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_j} L(y_i, f_{m-1}(x_i) + \gamma)$ - Update: $f_m(x) = f_{m-1}(x) + \nu \sum_j \gamma_{jm} I(x \in R_{jm})$

The parameter ν is called the **learning rate** or **shrinkage parameter**.

Modern Implementations

Gradient boosting has evolved into several highly-optimized implementations:

XGBoost (Extreme Gradient Boosting)

- Uses second-order Taylor approximation for faster convergence
- Built-in regularization on tree complexity
- Handles missing values automatically
- Parallel tree construction
- Very popular in competitions!

LightGBM

- **Gradient-based One-Side Sampling (GOSS)**: Focus on high-gradient (hard) examples
- **Exclusive Feature Bundling (EFB)**: Combine sparse features
- Leaf-wise tree growth (can be faster but risks overfitting)

CatBoost

- Superior handling of categorical features (ordered boosting)
- Reduces target leakage
- Often requires less tuning

Regularization and Tuning

Gradient boosting can overfit! Key regularization techniques:

1. Shrinkage (Learning Rate)

Scale each update by $\nu \in (0, 1)$:

$$f_m(x) = f_{m-1}(x) + \nu \cdot h_m(x)$$

Smaller ν: - Requires more trees (more iterations) - But generalizes better! - Typical values: 0.01 to 0.1

2. Subsampling

Train each tree on a random subset of the training data: - Reduces variance - Speeds up training - Typical values: 50-80% of data

3. Early Stopping

Monitor performance on a validation set. Stop when validation error stops improving: - Prevents overfitting - Saves computation - Most practical stopping criterion

4. Tree Constraints

Limit tree complexity: - **Max depth**: Typically 3-8 - **Min samples per leaf**: Prevents tiny leaves - **Max leaves**: Direct control on complexity

---

AdaBoost vs. Gradient Boosting

Aspect	AdaBoost	Gradient Boosting
Loss	Exponential only	Any differentiable
Weight update	Reweight observations	Fit pseudo-residuals
Robustness	Sensitive to outliers	Can use robust losses
Flexibility	Classification focus	Regression and classification
Historical	First boosting algorithm	Modern standard

Summary

Key Concepts

1. **Weak learners + boosting = strong learner**: The magic of combining simple models
2. **Sequential focusing**: Each round corrects previous mistakes
3. **Gradient descent in function space**: Gradient boosting's unifying principle
4. **Regularization is crucial**: Learning rate, subsampling, early stopping
5. **Trees are the workhorses**: Gradient boosted trees dominate tabular data

When to Use Boosting

Great for: - Tabular data - When you need maximum accuracy - When you can tune hyperparameters

Not ideal for: - When interpretability is paramount - Very small datasets - When training time is extremely limited

Practical Tips

1. Start with a small learning rate (0.01-0.1)
2. Use early stopping based on validation error
3. Tune max_depth (3-8) and n_estimators together
4. Consider subsampling for large datasets
5. Try XGBoost, LightGBM, or CatBoost — they're all excellent!

ewpage

Random Forests

Random Forests are one of the most successful and widely-used machine learning algorithms. They combine the simplicity of decision trees with the power of ensemble methods to create a robust, accurate, and easy-to-use classifier.

The Big Picture

Decision trees are intuitive and interpretable, but they have a major flaw: **high variance**. A small change in the training data can produce a completely different tree. Random Forests solve this by building many trees and averaging their predictions.

Key insight: Many imperfect models, when combined intelligently, can outperform a single "perfect" model.

Why Averaging Helps

The Bias-Variance View

Individual decision trees (especially deep ones) have: - **Low bias**: They can fit complex patterns - **High variance**: They're sensitive to the specific training data

Averaging reduces variance while maintaining low bias!

Mathematical Intuition

Consider B random variables (predictions from B trees), each with: - Individual variance: σ^2 - Pairwise correlation: ρ

The variance of their average is:

$$\text{Var}\left(\frac{1}{B}\sum_{b=1}^{B} X_b\right) = \rho\sigma^2 + \frac{(1-\rho)}{B}\sigma^2$$

Two key insights:

1. **More trees (larger B) always helps**: The second term shrinks toward 0

2. **Lower correlation (ρ) helps even more**: The first term shrinks

This is why Random Forests can't overfit by adding more trees! (Unlike boosting, which can overfit with too many rounds.)

Bagging: The Foundation

Bagging (Bootstrap Aggregating) is the simpler ancestor of Random Forests.

The Algorithm

1. **Bootstrap**: Draw B samples of size N with replacement from training data
2. **Train**: Fit a decision tree to each bootstrap sample
3. **Aggregate**:
   - Regression: Average predictions
   - Classification: Majority vote

Why Bootstrap?

Each bootstrap sample is slightly different from the original data: - Contains ~63.2% of unique observations - Some observations appear multiple times - Others don't appear at all

This variation creates diversity among trees — different trees make different mistakes!

Random Forests: Adding More Randomness

Random Forests add an additional source of randomness to further **decorrelate** the trees.

The Key Innovation

At each split, instead of considering all features, consider only a **random subset** of m features.

Why this helps: - In bagging, if one feature is very strong, every tree uses it at the root □ trees are correlated - Random feature selection forces trees to use different features □ less correlation

Typical Values for m

Task	Recommended m
Classification	$\sqrt{p}$
Regression	$p/3$

Where p = total number of features.

The Complete Algorithm

1. For b = 1 to B trees:
   - Draw a bootstrap sample of size N
   - Grow a tree:
     - At each node, randomly select m features
     - Find the best split among those m features
     - Split the node
     - Repeat until stopping criterion (min node size)
2. Output: Ensemble of B trees

For prediction: - **Regression**: $\hat{f}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}_b(x)$ - **Classification**: $\hat{G}(x) =$ majority vote of $\hat{G}_b(x)$

Out-of-Bag (OOB) Error

One of the most elegant features of Random Forests: **free cross-validation**!

The Idea

Each bootstrap sample leaves out ~36.8% of observations. For each observation i: 1. Find all trees where i was NOT in the training sample 2. Use only those trees to predict for i 3. This is an honest prediction — no leakage!

OOB Error Estimate

$$\text{OOB Error} = \frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{y}_i^{\text{OOB}})$$

Where $\hat{y}_i^{\text{OOB}}$ is the prediction using only trees that didn't train on observation i.

Properties

- **Essentially equivalent to leave-one-out cross-validation**
- **Computed for free** during training
- **No need for separate validation set**
- **Honest estimate** of generalization error

Variable Importance

Random Forests provide built-in measures of which features matter most.

Method 1: Mean Decrease in Impurity

For each feature j: 1. At each split on feature j, record the decrease in impurity (Gini, entropy, or MSE) 2. Sum across all splits and all trees 3. Normalize

Pros: Fast, computed during training **Cons**: Biased toward high-cardinality categorical features

Method 2: Permutation Importance

A more reliable approach:

1. Compute OOB accuracy for the original data
2. For each feature j:
    - Randomly shuffle (permute) feature j's values
    - Recompute OOB accuracy
    - Record the decrease in accuracy
3. Average across all trees

Interpretation: If shuffling feature j destroys accuracy, that feature was important!

Pros: Unbiased, captures complex dependencies **Cons**: Computationally more expensive

When Variables Are Correlated

Both importance measures can be misleading with correlated features: - Importance may be split among correlated features - A single feature from a correlated group may show low importance

---

Proximity Measures

Random Forests can measure similarity between observations.

Computing Proximities

For each pair of observations (i, k): 1. Count how often they end up in the same terminal node 2. Across all trees 3. Normalize by number of trees

This creates an N × N **proximity matrix**.

Uses

- **Visualization**: Use multidimensional scaling (MDS) to plot proximities in 2D
- **Outlier detection**: Points with low average proximity to their class may be outliers
- **Missing value imputation**: Fill in missing values using weighted averages of similar observations
- **Clustering**: Use proximity as a similarity measure

Advantages of Random Forests

1. Accuracy

- Often among the best performing methods “out of the box”
- Works well on many types of data without much tuning

2. Robustness

- Handles missing values gracefully
- Not sensitive to outliers (median of trees is robust)
- Works with both categorical and continuous features

3. Scalability

- Parallelizes naturally (trees are independent)
- Handles large datasets efficiently

4. Interpretability (relative to other ensembles)

- Variable importance provides insights
- Individual trees can be examined
- Proximities enable visualization

5. Built-in Validation

- OOB error provides honest generalization estimate
- No need for separate cross-validation

Limitations

1. Less Interpretable Than Single Trees

- Can’t see a single “path” to a prediction
- Hard to extract simple rules

2. Memory and Speed

- Storing many trees requires memory
- Prediction time scales with number of trees

3. Extrapolation

- Can’t extrapolate beyond the range of training data
- Predictions for extreme values will be bounded by training range

4. Imbalanced Classes

- May be biased toward majority class
- May need class weights or sampling adjustments

Hyperparameter Tuning

Random Forests have relatively few hyperparameters:

Parameter	Description	Typical Values	Effect
n_estimators	Number of trees	100-1000	More is better (diminishing returns)
max_features	Features per split	sqrt(p), p/3	Lower □ more diversity, more variance
max_depth	Tree depth	None (grow full), or 10-30	Deeper □ lower bias, higher variance
min_samples_leaf	Min samples in leaf	1-10	Higher □ smoother, simpler trees

Practical Tips

1. **Start with defaults**: They often work well!
2. **More trees rarely hurt**: Just costs compute time
3. **max_features is most important**: Try sqrt(p), log(p), and p/3
4. **Deeper trees are fine**: Unlike single trees, Random Forests resist overfitting

---

Random Forests vs. Boosting

Aspect	Random Forests	Gradient Boosting
Training	Parallel (independent trees)	Sequential (each tree corrects errors)
Overfitting	Very resistant	Can overfit with too many rounds
Tuning	Minimal tuning needed	Requires careful tuning
Accuracy	Very good	Often slightly better (with tuning)
Speed	Fast (parallelizable)	Slower (sequential)
Interpretability	Variable importance	Less interpretable

When to Use Which?

Random Forests: - Quick baseline - Limited tuning time - Parallel computing available - Want OOB error estimate

Gradient Boosting: - Maximum accuracy needed - Time for hyperparameter tuning - Tabular data competitions

---

Summary

Key Takeaways

1. **Averaging reduces variance**: The fundamental principle behind Random Forests
2. **Decorrelation is key**: Random feature selection makes trees diverse
3. **Can't overfit by adding trees**: Unlike many methods, more trees is (almost) always better
4. **OOB error is free cross-validation**: Built-in generalization estimate
5. **Variable importance provides insights**: Understand which features matter
6. **Minimal tuning required**: Works well out of the box

The Random Forest Workflow

```
1. Train Random Forest with default settings
2. Check OOB error for baseline performance
3. Examine variable importance for insights
4. If needed, tune max_features and min_samples_leaf
5. For final model, use more trees (500-1000)
```

Random Forests remain one of the best "first try" algorithms for tabular data — accurate, robust, and easy to use!

ewpage

Part III: Probabilistic Machine Learning Notes

ewpage

Probabilistic Machine Learning Notes

These notes are based on "Probabilistic Machine Learning" by Kevin Murphy — a comprehensive modern treatment of machine learning from a probabilistic perspective. This guide makes these concepts accessible to undergraduates while maintaining the depth needed for practical understanding.

What is Probabilistic Machine Learning?

Probabilistic ML treats machine learning as a problem of **inference under uncertainty**. Instead of just making predictions, we quantify how confident we are in those predictions. This is crucial for real-world applications where decisions have consequences.

Key idea: Everything is uncertain — our data is noisy, our models are approximations, and we never have enough data. Probability theory gives us a principled framework to reason about this uncertainty.

Why the Probabilistic Perspective?

1. **Quantified Uncertainty**: Know when to trust your model's predictions
2. **Principled Learning**: Derive optimal learning algorithms from first principles
3. **Regularization**: Prevent overfitting through priors and Bayesian inference
4. **Model Comparison**: Compare different models in a principled way
5. **Decision Making**: Make optimal decisions under uncertainty

Topics Covered

Topic	What You'll Learn
Probability	Foundation of uncertainty quantification
Statistics	Inference, estimation, and hypothesis testing
Decision Theory	Making optimal choices under uncertainty
Information Theory	Measuring information and uncertainty
Optimization	Finding the best model parameters
Discriminant Analysis	Generative vs. discriminative models
Linear & Logistic Regression	Foundational supervised learning

Topic	What You'll Learn
Neural Networks	Deep learning architectures (FFN, CNN, RNN)
Trees & Ensembles	Decision trees, random forests, boosting
Exemplar Methods	KNN, metric learning
Self-Supervised Learning	Learning from unlabeled data
Recommendation Systems	Collaborative filtering and matrix factorization

Prerequisites

To get the most from these notes: - **Calculus**: Derivatives, gradients, chain rule - **Linear Algebra**: Matrices, eigenvalues, matrix decompositions - **Basic Probability**: Random variables, expectations, common distributions - **Programming**: Python with NumPy, familiarity with ML libraries helpful

How to Use These Notes

1. **Start with foundations**: Probability and statistics chapters build the foundation
2. **Understand the "why"**: Focus on intuition before equations
3. **Connect concepts**: Many ideas recur across chapters (e.g., MLE, regularization)
4. **Practice**: Implement algorithms to solidify understanding

Let's dive in!

ewpage

Introduction to Machine Learning

Machine learning is the science of getting computers to learn from data without being explicitly programmed. This chapter introduces the fundamental concepts, problem types, and challenges that define the field.

What is Machine Learning?

Tom Mitchell's Definition: > A computer program is said to learn from experience E with respect to some class of tasks T, and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

In plain English: A machine learning system gets better at a task as it sees more data.

Example: A spam filter - **Task (T)**: Classify emails as spam or not spam - **Experience (E)**: A dataset of labeled emails - **Performance (P)**: Accuracy on new emails

---

Supervised Learning

In supervised learning, we have input-output pairs and want to learn the mapping between them.

The Setup

- **Inputs** (X): Also called features, covariates, or predictors
  - Example: Pixel values of an image, words in an email
- **Outputs** (Y): Also called labels, targets, or responses
  - Example: "cat" or "dog", spam or not spam

Goal: Learn a function $f : X \to Y$ that generalizes to new, unseen examples.

Classification

When the output is a **discrete category**:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} I\{y_i \neq f(x_i, \theta)\}$$

This is the **misclassification rate** — the fraction of examples we get wrong.

Key concepts: - **Empirical Risk**: Average loss on training data - **Empirical Risk Minimization (ERM)**: Find parameters that minimize training loss - **Generalization**: The real goal is to perform well on *new* data, not just training data

Dealing with Uncertainty

Models can't predict with 100% certainty. There are two types of uncertainty:

Model Uncertainty (Epistemic) - Arises from lack of knowledge about the true mapping - Can be reduced with more data - Example: We don't know if a blurry image is a cat or dog

Data Uncertainty (Aleatoric) - Arises from inherent randomness in the data - Cannot be reduced even with infinite data - Example: A coin flip is inherently random

Probabilistic Predictions

Instead of just predicting a class, predict a probability distribution over classes:

$$p(y|x, \theta)$$

Why probabilities? - Quantify confidence - Enable better decision making - Allow principled handling of uncertainty

Negative Log-Likelihood (NLL):

$$\text{NLL}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log p(y_i|f(x_i, \theta))$$

Minimizing NLL is equivalent to **Maximum Likelihood Estimation (MLE)** — finding parameters that make the observed data most probable.

Regression

When the output is a **continuous value**:

$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(x_i, \theta))^2$$

This is **Mean Squared Error (MSE)** — the average squared difference between predictions and true values.

Connection to Probability: If we assume Gaussian noise:

$$p(y|x, \theta) = \mathcal{N}(y|f(x, \theta), \sigma^2)$$

Then minimizing NLL is equivalent to minimizing MSE!

Types of Regression Models

Model	Description	Flexibility
Linear	$f(x) = w^T x + b$	Low
Polynomial	Includes x^2, x^3, etc.	Medium
Neural Networks	Nested nonlinear functions	High

Overfitting and Generalization

The Overfitting Problem

A model that perfectly fits training data but fails on new data is **overfitting**. It has memorized the training set rather than learning the underlying pattern.

Signs of overfitting: - Training error much lower than test error - Model is very complex relative to data size - Model captures noise as if it were signal

Understanding the Errors

Population Risk: Theoretical expected loss on the true data generating process

$$R(\theta) = \mathbb{E}_{(x,y)\sim p^*}[L(y, f(x, \theta))]$$

Empirical Risk: Average loss on training data

$$\hat{R}(\theta) = \frac{1}{N}\sum_{i=1}^{N} L(y_i, f(x_i, \theta))$$

Generalization Gap: Difference between population and empirical risk

$$\text{Gap} = R(\theta) - \hat{R}(\theta)$$

A large gap indicates overfitting.

The U-Shaped Test Error Curve

```
Error

                        Training Error
            ___________ (keeps decreasing)
       ____

                Test
```

```
      Error        (U-shaped)

           → Model Complexity
Simple          Complex
```

- **Underfitting** (left): Model too simple, high bias
- **Sweet spot** (middle): Good balance
- **Overfitting** (right): Model too complex, high variance

No Free Lunch Theorem

There is no single best model that works for all problems.

Every model makes assumptions about the data. When those assumptions match reality, the model works well. When they don't, it fails.

Implication: Understanding your problem domain is crucial for choosing the right model.

Unsupervised Learning

In unsupervised learning, we only have inputs X — no labels.

Goal: Discover hidden structure in data.

Common Tasks

Clustering: Group similar data points together - Example: Customer segmentation, document grouping

Dimensionality Reduction: Find lower-dimensional representations - Example: Compress images while preserving important information

Density Estimation: Model the probability distribution $p(x)$ - Example: Anomaly detection (low probability = anomalous)

Self-Supervised Learning: Create proxy tasks from unlabeled data - Example: Predict missing words in text (BERT), predict next frame in video

Evaluation Challenge

Without labels, how do we evaluate? Common approaches: - Likelihood of held-out data - Performance on downstream tasks - Human evaluation of quality

---

Reinforcement Learning

An **agent** learns to interact with an **environment** to maximize cumulative **reward**.

Key differences from supervised learning: - No explicit labels — only reward signals - Rewards are often delayed (sparse feedback) - Agent's actions affect future states (sequential decision making)

Analogy: - Supervised learning = learning with a teacher who gives correct answers - Reinforcement learning = learning with a critic who only says "good" or "bad"

Data Preprocessing

Text Data

Raw text needs transformation before ML models can process it.

Bag of Words (BoW) - Represent document as vector of word counts - Loses word order but captures content

Problem: Common words ("the", "a") dominate counts.

TF-IDF (Term Frequency - Inverse Document Frequency):

$$\text{TF-IDF} = \log(1 + \text{TF}) \times \text{IDF}$$

Where: - TF = term frequency (how often word appears in document) - IDF = log $\frac{N}{1+\text{DF}}$ (inverse of how many documents contain the word)

Effect: Downweight common words, upweight distinctive words.

Word Embeddings: Map words to dense vectors that capture semantic meaning - Similar words have similar vectors - "king" - "man" + "woman" ≈ "queen"

Handling Unknown Words: - UNK token: Replace rare/unseen words with a special token - Subword units (BPE): Break words into common pieces

Missing Data

How data is missing matters!

Type	Description	Example	Handling
MCAR	Missing Completely At Random	Random sensor failures	Easier to handle
MAR	Missing At Random (depends on observed data)	Older people less likely to report income	Model the missingness
NMAR	Not Missing At Random	Sick people skip health surveys	Most challenging

Summary

Concept	Key Insight
ML Definition	Learning improves with experience
Supervised Learning	Learn input-output mapping from labeled data
Classification	Predict discrete categories
Regression	Predict continuous values
Probabilistic View	Quantify uncertainty with probability distributions
Overfitting	Memorizing training data instead of learning patterns
Generalization	The ultimate goal: perform well on new data
Unsupervised Learning	Find structure without labels
RL	Learn from reward signals through interaction

The probabilistic perspective unifies these concepts — learning is inference under uncertainty.

ewpage

Probability Foundations

Probability is the mathematical language of uncertainty. In machine learning, it provides the foundation for reasoning about noisy data, model uncertainty, and making predictions. This chapter covers the essential probability concepts you'll use throughout ML.

Two Views of Probability

Frequentist View

Probability is the **long-run relative frequency** of an event in repeated experiments.

Example: The probability of heads is 0.5 because if you flip a coin many times, about half will be heads.

Limitation: What about events that can't be repeated? (e.g., "probability it rains tomorrow")

Bayesian View

Probability is a **quantification of subjective uncertainty** or degree of belief.

Example: "I'm 70% confident it will rain tomorrow" represents my current belief given available evidence.

Advantage: Can express uncertainty about any proposition, including model parameters.

Two Types of Uncertainty

Epistemic Uncertainty (Model Uncertainty) - Uncertainty due to lack of knowledge - Can be reduced with more data - Example: Uncertainty about which model is correct

Aleatoric Uncertainty (Data Uncertainty) - Uncertainty due to inherent randomness - Cannot be reduced even with infinite data - Example: Outcome of a fair coin flip

---

Basic Probability Rules

Events and Probabilities

An **event** A is some state of the world that either holds or doesn't.

Probability axioms: - $0 \leq P(A) \leq 1$ (probabilities are between 0 and 1) - $P(A) + P(\bar{A}) = 1$ (something happens or it doesn't) - $P(\Omega) = 1$ (something must happen)

Joint Probability

The probability that **both** A and B occur:

$$P(A, B) = P(A \cap B)$$

Special case — Independence: If A and B are independent:

$$P(A, B) = P(A) \cdot P(B)$$

Inclusion-Exclusion Principle:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Conditional Probability

The probability of B **given that** A has occurred:

$$P(B|A) = \frac{P(A \cap B)}{P(A)} \quad \text{where } P(A) > 0$$

Intuition: We restrict our attention to the "world where A happened" and ask how likely B is in that world.

Example: P(has cancer I positive test) ≠ P(positive test I has cancer)

This is a common source of confusion called the **base rate fallacy**!

Random Variables

A **random variable** is a function that maps outcomes from a sample space to real numbers. This allows mathematical manipulation of random events.

Discrete Random Variables

Take values from a countable set (integers, categories). - Example: Number of customers, dice roll, coin flip

Continuous Random Variables

Take values from an uncountable set (real numbers, intervals). - Example: Height, temperature, time

---

Probability Distributions

Probability Mass Function (PMF)

For discrete random variables, the PMF gives the probability of each value:

$$p(x) = P(X = x)$$

Properties: - $0 \leq p(x) \leq 1$ for all x - $\sum_x p(x) = 1$ (probabilities sum to 1)

Probability Density Function (PDF)

For continuous random variables, the PDF describes relative likelihood:

$$P(a \leq X \leq b) = \int_a^b f(x)dx$$

Important: For continuous variables, $P(X = x) = 0$ for any specific x!

Properties: - $f(x) \geq 0$ (but can be greater than 1!) - $\int_{-\infty}^{\infty} f(x)dx = 1$

Cumulative Distribution Function (CDF)

The probability that X is less than or equal to x:

$$F_X(x) = P(X \leq x)$$

Properties: - Monotonically non-decreasing - $\lim_{x \to -\infty} F_X(x) = 0$ - $\lim_{x \to \infty} F_X(x) = 1$ - $P(a \leq X \leq b) = F_X(b) - F_X(a)$

Relationship: PDF is the derivative of CDF.

Inverse CDF (Quantile Function)

Given a probability, find the value:

$$F^{-1}(q) = \inf\{x : F(x) \geq q\}$$

Common quantiles: - $F^{-1}(0.5)$ = median - $F^{-1}(0.25)$, $F^{-1}(0.75)$ = lower and upper quartiles

---

Working with Multiple Variables

Marginal Distribution

Given joint distribution $p(X, Y)$, the marginal distribution of X:

$$p(X = x) = \sum_y p(X = x, Y = y)$$

Intuition: “Sum out” the variable you don’t care about.

Conditional Distribution

$$p(Y = y|X = x) = \frac{p(X = x, Y = y)}{p(X = x)}$$

Product Rule

$$p(X, Y) = p(Y|X) \cdot p(X) = p(X|Y) \cdot p(Y)$$

Chain Rule

For multiple variables:

$$p(X_1, X_2, X_3) = p(X_1) \cdot p(X_2|X_1) \cdot p(X_3|X_1, X_2)$$

Independence

X and Y are **independent** if:

$$X \perp Y \Leftrightarrow p(X, Y) = p(X) \cdot p(Y)$$

Equivalently: Knowing X tells you nothing about Y.

Conditional Independence

X and Y are **conditionally independent** given Z if:

$$X \perp Y|Z \Leftrightarrow p(X, Y|Z) = p(X|Z) \cdot p(Y|Z)$$

Example: Given the weather, whether I carry an umbrella is independent of whether you carry one. But marginally (without knowing the weather), they're correlated!

Summary Statistics

Expected Value (Mean)

The "center" of a distribution — the average value weighted by probability:

$$\mathbb{E}[X] = \sum_x x \cdot p(x) \quad \text{or} \quad \int_{-\infty}^{\infty} x \cdot f(x)dx$$

Linearity of Expectation (extremely useful!):

$$\mathbb{E}[aX + b] = a\mathbb{E}[X] + b$$

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y] \quad \text{(always, even if not independent!)}$$

Variance

How spread out the distribution is:

$$\text{Var}(X) = \mathbb{E}[(X - \mu)^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$$

Properties: - $\text{Var}(aX + b) = a^2\text{Var}(X)$ - $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X, Y)$

Mode

The most probable value:

$$\text{mode} = \arg\max_x p(x)$$

Laws of Iterated Expectations

Law of Total Expectation

$$\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X|Y]]$$

Intuition: The overall average equals the average of conditional averages.

Law of Total Variance

$$\text{Var}(X) = \mathbb{E}[\text{Var}(X|Y)] + \text{Var}(\mathbb{E}[X|Y])$$

Interpretation: - First term: Average variance within groups - Second term: Variance between group means

---

Bayes' Rule

The foundation of Bayesian inference:

$$P(H|Y) = \frac{P(Y|H) \cdot P(H)}{P(Y)}$$

Components: - $P(H)$: **Prior** — belief before seeing data - $P(Y|H)$: **Likelihood** — probability of data given hypothesis - $P(H|Y)$: **Posterior** — updated belief after seeing data - $P(Y)$: **Evidence** — normalizing constant

The Bayesian Recipe:

$$\text{Posterior} \propto \text{Prior} \times \text{Likelihood}$$

Common Distributions

Bernoulli Distribution

Models a single binary outcome:

$$Y \sim \text{Ber}(\theta)$$

$$p(Y = y) = \theta^y (1-\theta)^{1-y} \quad \text{for } y \in \{0, 1\}$$

- Mean: θ
- Variance: $\theta(1-\theta)$

Binomial Distribution

Models number of successes in N independent Bernoulli trials:

$$p(k|N, \theta) = \binom{N}{k} \theta^k (1-\theta)^{N-k}$$

- Mean: $N\theta$
- Variance: $N\theta(1-\theta)$

Logistic Function

The **sigmoid** function maps any real number to (0, 1):

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

Properties: - $\sigma(-a) = 1 - \sigma(a)$ - $\sigma'(a) = \sigma(a)(1 - \sigma(a))$

Log-odds (logit): The inverse transformation:

$$a = \log \frac{p}{1-p}$$

Usage: Binary classification — map raw scores to probabilities:

$$p(y = 1|x, \theta) = \sigma(f(x, \theta))$$

Categorical Distribution

Generalizes Bernoulli to multiple categories:

$$\text{Cat}(y|\theta) = \prod_{c=1}^{C} \theta_c^{I(y=c)}$$

Constraints: - $0 \leq \theta_c \leq 1$ - $\sum_c \theta_c = 1$

Softmax Function

Maps raw logits to valid categorical probabilities:

$$\text{softmax}(a)_c = \frac{e^{a_c}}{\sum_{j=1}^{C} e^{a_j}}$$

Properties: - Output sums to 1 - Each output is in (0, 1) - Invariant to adding constant to all inputs

Temperature Scaling: Divide by temperature T before softmax: - T □ 0: "Winner takes all" (approaches argmax) - T □ ∞: Approaches uniform distribution

Log-Sum-Exp Trick

For numerical stability when computing softmax:

$$\log \sum_c e^{a_c} = m + \log \sum_c e^{a_c - m}$$

where $m = \max_c a_c$. This prevents overflow!

Gaussian (Normal) Distribution

The workhorse of statistics:

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Properties: - Mean, median, and mode all equal μ - Symmetric around the mean - 68-95-99.7 rule: ~68% within 1σ, ~95% within 2σ, ~99.7% within 3σ

Why Gaussian is so common: 1. Maximum entropy distribution for given mean and variance 2. Central Limit Theorem: Sum of many independent RVs □ Gaussian 3. Mathematical convenience (conjugate to itself)

Student's t-Distribution

More robust alternative to Gaussian (heavier tails): - PDF decays polynomially, not exponentially - Parameter ν (degrees of freedom) controls tail weight - As ν □ ∞, approaches Gaussian - Better for handling outliers

---

Transformations of Random Variables

Discrete Case

If Y = f(X), the PMF of Y:

$$p_Y(y) = \sum_{x: f(x)=y} p_X(x)$$

Continuous Case (Change of Variables)

If Y = f(X) where f is monotonic and differentiable:

$$p_Y(y) = p_X(f^{-1}(y)) \cdot \left| \frac{df^{-1}(y)}{dy} \right|$$

The absolute value of the derivative (Jacobian in multivariate case) accounts for how the transformation stretches or compresses probability.

Convolution

For $Y = X_1 + X_2$ (sum of independent RVs):

$$p_Y(y) = \int p_{X_1}(x_1) \cdot p_{X_2}(y - x_1) dx_1$$

Key result: Sum of Gaussians is Gaussian!

Central Limit Theorem

One of the most important theorems in statistics:

If $X_1, X_2, ..., X_N$ are i.i.d. with mean μ and variance σ², then as N □ ∞:

$$\bar{X} = \frac{1}{N} \sum_{i=1}^{N} X_i \xrightarrow{d} \mathcal{N}\left(\mu, \frac{\sigma^2}{N}\right)$$

Implications: - Sample means are approximately Gaussian (for large N) - Justifies using Gaussian assumptions in many settings - Variance of sample mean decreases as 1/N

Monte Carlo Approximation

When exact computation is intractable, **sample**!

To estimate $\mathbb{E}[f(X)]$ where $X \sim p(x)$: 1. Draw samples $x_1, x_2, ..., x_N \sim p(x)$ 2. Approximate: $\mathbb{E}[f(X)] \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i)$

This is unbiased and converges by the Law of Large Numbers.

Summary

Concept	Key Formula	Intuition
Conditional Probability	$P(B\|A) = P(A,B)/P(A)$	Probability in restricted world
Bayes' Rule	$P(H\|Y) \propto P(Y\|H)P(H)$	Update beliefs with evidence
Expected Value	$\mathbb{E}[X] = \sum x \cdot p(x)$	Probability-weighted average
Variance	$\text{Var}(X) = \mathbb{E}[(X-\mu)^2]$	Spread of distribution
Sigmoid	$\sigma(a) = 1/(1+e^{-a})$	Map reals to (0,1)
Softmax	$e^{a_c}/\sum e^{a_j}$	Map vector to probabilities
CLT	$\bar{X} \to \mathcal{N}(\mu, \sigma^2/N)$	Sample means are Gaussian

ewpage

Probability: Advanced Topics

This chapter covers more advanced probability concepts including covariance, correlation, mixture models, and Markov chains — essential tools for understanding many machine learning algorithms.

Covariance and Correlation

Covariance

Covariance measures how two random variables **vary together**:

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$$

Interpretation: - **Positive covariance**: When X is above its mean, Y tends to be above its mean - **Negative covariance**: When X is above its mean, Y tends to be below its mean - **Zero covariance**: No linear relationship

Alternative formula:

$$\text{Cov}(X, Y) = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$$

Properties: - $\text{Cov}(X, X) = \text{Var}(X)$ - $\text{Cov}(X, Y) = \text{Cov}(Y, X)$ - $\text{Cov}(aX + b, Y) = a \cdot \text{Cov}(X, Y)$

Correlation

Correlation is a **scaled** version of covariance, always between -1 and 1:

$$\rho_{XY} = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)} \cdot \sqrt{\text{Var}(Y)}}$$

Interpretation: - $\rho = 1$: Perfect positive linear relationship - $\rho = -1$: Perfect negative linear relationship - $\rho = 0$: No linear relationship (but could have non-linear relationship!)

Independence vs. Uncorrelation

Key distinction: - **Independent □ Uncorrelated** (always true) - **Uncorrelated □ NOT □ Independent** (converse is false!)

Example: Let X ~ Uniform(-1, 1) and Y = X². Then: - Cov$(X, Y) = 0$ (by symmetry) - But X and Y are clearly dependent (Y is completely determined by X!)

Correlation ≠ Causation

A correlation between X and Y could be due to: 1. X causes Y 2. Y causes X 3. A third variable Z causes both (confounding) 4. Pure coincidence (spurious correlation)

Simpson's Paradox: A trend that appears in groups of data can **disappear or reverse** when groups are combined. Always be cautious about aggregated data!

Mixture Models

Mixture models represent complex distributions as **weighted combinations** of simpler distributions.

Definition

$$p(y|\theta) = \sum_{k=1}^{K} \pi_k \cdot p_k(y)$$

Where: - π_k are **mixing proportions** (weights): $\sum_k \pi_k = 1$, all $\pi_k \geq 0$ - $p_k(y)$ are **component distributions**

Generative Process

To sample from a mixture: 1. Sample component index: $k \sim$ Categorical$(\pi_1, ..., \pi_K)$ 2. Sample from chosen component: $y \sim p_k(y)$

Gaussian Mixture Models (GMMs)

The most common mixture model uses Gaussian components:

$$p(y) = \sum_{k=1}^{K} \pi_k \cdot \mathcal{N}(y|\mu_k, \Sigma_k)$$

Applications: - **Clustering**: Soft assignment of points to clusters - **Density estimation**: Model complex, multi-modal distributions - **Outlier detection**: Points with low probability under all components

K-Means as a Special Case

K-Means clustering is a limiting case of GMM with: - Uniform mixing proportions: $\pi_k = 1/K$ - Spherical Gaussians: $\Sigma_k = \sigma^2 I$ (same for all components) - Hard assignments (as $\sigma \to 0$)

Markov Chains

Markov chains model **sequences** where each state depends only on the previous state.

Chain Rule for Sequences

For a sequence $(x_1, x_2, x_3, ...)$:

$$p(x_1, x_2, x_3) = p(x_1) \cdot p(x_2|x_1) \cdot p(x_3|x_1, x_2)$$

This is exact but requires modeling complex conditional dependencies.

The Markov Assumption

First-order Markov property: The future depends only on the present, not the past.

$$p(x_t|x_1, x_2, ..., x_{t-1}) = p(x_t|x_{t-1})$$

This simplifies the chain rule to:

$$p(x_1, x_2, x_3) = p(x_1) \cdot p(x_2|x_1) \cdot p(x_3|x_2)$$

State Transition Matrix

The conditional distribution $p(x_t|x_{t-1})$ defines a **transition matrix** T where:

$$T_{ij} = p(x_t = j|x_{t-1} = i)$$

Properties: - Rows sum to 1 (each row is a valid probability distribution) - T^n gives n-step transition probabilities

Higher-Order Markov Models

Second-order Markov: $p(x_t|x_{t-1}, x_{t-2})$ - Used in bigram language models

n-th order Markov: $p(x_t|x_{t-1}, ..., x_{t-n})$ - Used in n-gram language models

Trade-off: Higher order captures more context but requires more parameters.

Applications in ML

- **Language modeling**: Predicting the next word
- **Hidden Markov Models**: Markov chains with hidden states
- **Reinforcement learning**: MDP (Markov Decision Process)
- **MCMC**: Markov Chain Monte Carlo for sampling

The Multivariate Gaussian

The multivariate Gaussian (MVN) is crucial for modeling correlated continuous variables.

Definition

For a d-dimensional random vector **x**:

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

Where: - μ: Mean vector (d × 1) - Σ: Covariance matrix (d × d, symmetric positive definite)

Key Properties

Marginals: If $(X, Y) \sim \mathcal{N}$, then $X \sim \mathcal{N}$ and $Y \sim \mathcal{N}$

Conditionals: $X|Y \sim \mathcal{N}$ (also Gaussian!)

Linear transformations: If **x** $\sim \mathcal{N}(\mu, \Sigma)$, then:

$$A\mathbf{x} + \mathbf{b} \sim \mathcal{N}(A\mu + \mathbf{b}, A\Sigma A^T)$$

Geometry of the Covariance Matrix

The covariance matrix determines the shape of the Gaussian: - **Diagonal Σ**: Ellipse aligned with axes - **Spherical Σ = σ²I**: Circle/sphere - **General Σ**: Rotated ellipse

The eigenvectors of Σ give the principal axes; eigenvalues give the variance along each axis.

Summary

Concept	Key Insight
Covariance	Measures linear co-variation; unscaled
Correlation	Scaled covariance between -1 and 1
Independence	Implies zero correlation, but not vice versa
Mixture Models	Complex distributions as weighted sums of simple ones
GMM	Gaussian components; soft clustering
Markov Chains	Future depends only on present, not past
Transition Matrix	Encodes all transition probabilities
Multivariate Gaussian	Generalizes Gaussian to multiple correlated variables

ewpage

Statistics

Statistics is the science of learning from data. This chapter covers the key concepts for estimating model parameters and quantifying uncertainty in those estimates.

The Big Picture

Inference: Quantifying uncertainty about unknown quantities using finite data samples.

Two major paradigms: - **Frequentist**: Parameters are fixed; uncertainty comes from random sampling - **Bayesian**: Parameters are random variables with prior distributions

---

Maximum Likelihood Estimation (MLE)

The most common approach to parameter estimation: choose parameters that make the observed data most probable.

The Setup

Given: - Data: $D = \{x_1, x_2, ..., x_N\}$ (assumed i.i.d.) - Parametric model: $p(x|\theta)$

The Likelihood Function

$$L(\theta; D) = p(D|\theta) = \prod_{i=1}^{N} p(x_i|\theta)$$

Key insight: We treat the data as fixed and vary θ. For which θ was this data most likely?

Log-Likelihood

Products are numerically unstable. Convert to sums using logs:

$$\ell(\theta; D) = \log L(\theta; D) = \sum_{i=1}^{N} \log p(x_i|\theta)$$

The MLE Estimate

$$\hat{\theta}_{MLE} = \arg\max_{\theta} \ell(\theta; D) = \arg\min_{\theta} -\ell(\theta; D)$$

Equivalently: minimize the **Negative Log-Likelihood (NLL)**.

Why MLE Works

Theoretical justifications: 1. **Bayesian view**: MLE is MAP estimate with uniform (uninformative) prior 2. **Information-theoretic view**: MLE minimizes KL divergence between model and empirical distribution

Sufficient Statistics

A **sufficient statistic** summarizes all information in the data relevant to estimating θ.

Example (Bernoulli): For N coin flips, the sufficient statistics are: - N_1 = number of heads - N_0 = number of tails

You don't need to know the order of the flips!

MLE Examples

Bernoulli Distribution

Model: $p(y|\theta) = \theta^y(1-\theta)^{1-y}$

NLL:

$$\text{NLL}(\theta) = -[N_1 \log \theta + N_0 \log(1-\theta)]$$

Setting derivative to zero:

$$\hat{\theta}_{MLE} = \frac{N_1}{N_0 + N_1} = \frac{\text{heads}}{\text{flips}}$$

Intuitive result: Estimate probability as observed frequency.

Gaussian Distribution

Model: $p(y|\mu, \sigma^2) = \mathcal{N}(y|\mu, \sigma^2)$

MLE estimates:

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} y_i \quad \text{(sample mean)}$$

$$\hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{\mu})^2 \quad \text{(sample variance)}$$

Linear Regression

Model: $p(y|x, w, \sigma^2) = \mathcal{N}(y|w^T x + b, \sigma^2)$

NLL is proportional to:

$$\text{NLL} \propto \sum_{i=1}^{N} (y_i - w^T x_i - b)^2$$

Key insight: MLE for Gaussian regression = minimize squared error!

Empirical Risk Minimization

ERM generalizes MLE beyond log-loss to any loss function.

Definition

$$\hat{\theta}_{ERM} = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, f(x_i; \theta))$$

Common loss functions: - Log-loss: gives MLE - Squared loss: regression - 0-1 loss: classification accuracy - Hinge loss: SVMs

Surrogate Losses

The 0-1 loss is non-differentiable. We use smooth **surrogate losses** that are easier to optimize: - Log-loss (cross-entropy) - Hinge loss - Exponential loss

---

Online Learning

When data arrives sequentially, we can't afford to retrain from scratch each time.

Recursive Updates

Many statistics can be updated incrementally:

Running mean:

$$\mu_t = \mu_{t-1} + \frac{1}{t}(x_t - \mu_{t-1})$$

Exponentially Weighted Moving Average (EWMA):

$$\mu_t = \beta \mu_{t-1} + (1 - \beta) x_t$$

Regularization

MLE can overfit — the estimated model fits training data perfectly but fails on new data.

The Problem

- Empirical distribution ≠ true distribution
- MLE finds parameters optimal for the empirical distribution
- May not generalize well

The Solution: Add a Penalty

$$\hat{\theta} = \arg\min_{\theta} \left[\text{NLL}(\theta) + \lambda R(\theta)\right]$$

Where $R(\theta)$ penalizes complex models.

MAP Estimation

From the Bayesian view, regularization corresponds to adding a prior:

$$\hat{\theta}_{MAP} = \arg\max_{\theta} [p(D|\theta) \cdot p(\theta)]$$

Taking logs:

$$\hat{\theta}_{MAP} = \arg\min_{\theta} [-\log p(D|\theta) - \log p(\theta)]$$

Examples: - Gaussian prior □ L2 regularization (Ridge) - Laplace prior □ L1 regularization (Lasso)

Choosing Regularization Strength

The regularization parameter λ controls the bias-variance trade-off.

Methods to choose λ: - **Validation set**: Test on held-out data - **Cross-validation**: For small datasets - **One Standard Error Rule**: Choose simplest model within one SE of best

Early Stopping

Another form of regularization: stop training before the model overfits. - Monitor validation error - Stop when it starts increasing

---

Bayesian Statistics

The Bayesian approach treats parameters as random variables.

The Bayesian Recipe

1. **Prior**: $p(\theta)$ — initial beliefs before seeing data
2. **Likelihood**: $p(D|\theta)$ — probability of data given parameters
3. **Posterior**: $p(\theta|D) \propto p(D|\theta) \cdot p(\theta)$ — updated beliefs

Posterior Predictive Distribution

To predict new data, **integrate over parameter uncertainty**:

$$p(y_{new}|x_{new}, D) = \int p(y_{new}|x_{new}, \theta) \cdot p(\theta|D) d\theta$$

Compare to plug-in prediction: $p(y_{new}|x_{new}, \hat{\theta})$

The Bayesian approach properly accounts for uncertainty in θ!

Conjugate Priors

When the prior and posterior have the same functional form: - **Bernoulli-Beta**: Prior on coin bias - **Gaussian-Gaussian**: Prior on Gaussian mean - **Poisson-Gamma**: Prior on Poisson rate

Makes computation tractable.

MAP vs. Full Bayesian

Aspect	MAP	Full Bayesian
Output	Point estimate	Full distribution
Computation	Optimization	Integration
Uncertainty	Not captured	Fully captured
Regularization	Equivalent to adding prior	Built-in

Frequentist Statistics

In the frequentist view: - Parameters θ are fixed (unknown) constants - Data D is random (sampled from true distribution) - Uncertainty comes from randomness in sampling

Sampling Distribution

If we repeated the experiment many times, our estimate $\hat{\theta}$ would vary. The **sampling distribution** describes this variation.

Bootstrap

When the true sampling distribution is unknown, approximate it by resampling:

1. Draw N samples **with replacement** from your data
2. Compute the statistic of interest
3. Repeat many times
4. The distribution of statistics approximates the sampling distribution

Key fact: Each bootstrap sample contains ~63.2% of unique original observations:

$$P(\text{included}) = 1 - \left(1 - \frac{1}{N}\right)^N \approx 1 - \frac{1}{e} \approx 0.632$$

Confidence Intervals

A 95% confidence interval means: if we repeated the experiment many times, 95% of the computed intervals would contain the true parameter.

Note: This is NOT the same as "95% probability that θ is in this interval"!

Bias-Variance Trade-off

Bias

How far off is our estimator on average?

$$\text{bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta^*$$

Unbiased: $\mathbb{E}[\hat{\theta}] = \theta^*$

Example: Sample variance $\frac{1}{N}\sum(x_i - \bar{x})^2$ is biased! The unbiased version divides by N-1.

Variance

How much does our estimate fluctuate across different datasets?

$$\text{Var}(\hat{\theta}) = \mathbb{E}[(\hat{\theta} - \mathbb{E}[\hat{\theta}])^2]$$

Mean Squared Error

Combines both:

$$\text{MSE}(\hat{\theta}) = \mathbb{E}[(\hat{\theta} - \theta^*)^2] = \text{bias}^2 + \text{variance}$$

Key insight: Sometimes it's worth accepting bias if it substantially reduces variance!

This is exactly what regularization does.

Summary

Concept	Key Idea
MLE	Choose θ that maximizes probability of observed data
NLL	Negative log-likelihood; what we minimize
Sufficient Statistics	Compress data without losing information about θ
ERM	Generalization of MLE to any loss function
Regularization	Penalty on complexity to prevent overfitting
MAP	MLE + prior = regularized MLE
Bayesian	Full distribution over θ, not just point estimate
Posterior Predictive	Integrate predictions over parameter uncertainty
Bootstrap	Approximate sampling distribution by resampling
Bias-Variance	MSE = bias² + variance; trade-off is fundamental

ewpage

Decision Theory

Decision theory provides a formal framework for making optimal choices under uncertainty. It bridges the gap between probabilistic predictions and concrete actions.

The Big Picture

Having a good model isn't enough — you need to **make decisions**. Decision theory tells us how to choose actions that minimize expected loss (or maximize expected utility).

Key question: Given uncertainty about the world and different costs for different errors, what should we do?

Risk Attitudes

Risk Neutrality

A risk-neutral agent values expected outcomes: - $50 for sure = 50% chance of $100

Risk Aversion

A risk-averse agent prefers certainty: - Would take $45 for sure over 50% chance of $100 - Most people are risk-averse for gains

Risk Seeking

A risk-seeking agent prefers uncertainty: - Would reject $55 for sure to keep 50% chance of $100 - Gamblers exhibit this behavior

---

Classification Decision Rules

Zero-One Loss

The simplest loss: you're either right or wrong.

$$\ell_{01}(y, \hat{y}) = I\{y \neq \hat{y}\} = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{if } y \neq \hat{y} \end{cases}$$

Optimal policy: Predict the most probable class!

$$\pi^*(x) = \arg\max_c P(Y = c|X = x)$$

Derivation: The risk (expected loss) for predicting $\hat{y}$:

$$R(\hat{y}|x) = P(Y \neq \hat{y}|x) = 1 - P(Y = \hat{y}|x)$$

Minimizing risk = maximizing the probability of being correct.

Cost-Sensitive Classification

Different errors have different consequences!

Medical diagnosis example: - **False Negative** (miss cancer): Potentially fatal - **False Positive** (false alarm): Unnecessary tests, anxiety

We assign different costs: - ℓ_{01}: Cost of predicting 0 when truth is 1 (false negative) - ℓ_{10}: Cost of predicting 1 when truth is 0 (false positive)

Optimal decision rule: Predict class 1 if:

$$P(Y = 0|x) \cdot \ell_{01} < P(Y = 1|x) \cdot \ell_{10}$$

Effect: Higher cost of false negatives lower threshold for predicting positive.

The Confusion Matrix

For binary classification:

	Predicted Positive	Predicted Negative
Actually Positive	True Positive (TP)	False Negative (FN)
Actually Negative	False Positive (FP)	True Negative (TN)

Key metrics:

Metric	Formula	Meaning
Sensitivity (Recall, TPR)	TP / (TP + FN)	Of actual positives, how many did we catch?
Specificity (TNR)	TN / (TN + FP)	Of actual negatives, how many did we correctly identify?

Metric	Formula	Meaning
Precision (PPV)	TP / (TP + FP)	Of predicted positives, how many are correct?
False Positive Rate (FPR)	FP / (FP + TN)	Of actual negatives, how many did we incorrectly flag?

The Rejection Option

Sometimes the best decision is to **not decide**.

Setup: - Cost of error: λ_e - Cost of rejection: λ_r (where $\lambda_r < \lambda_e$)

Optimal policy: - Predict if confident: $\max_c P(Y = c|x) \geq 1 - \frac{\lambda_r}{\lambda_e}$ - Reject (abstain) if uncertain

Use case: Route uncertain cases to human experts.

ROC Curves

The **Receiver Operating Characteristic** curve shows the trade-off between sensitivity and specificity across all classification thresholds.

Construction

For each threshold τ: 1. Compute TPR (sensitivity) and FPR (1 - specificity) 2. Plot the point (FPR, TPR)

Interpretation

- **Perfect classifier**: Goes through (0, 1) — 100% TPR, 0% FPR
- **Random classifier**: Diagonal line from (0, 0) to (1, 1)
- **Better models**: Curves closer to upper-left corner

AUC (Area Under the ROC Curve)

A single number summarizing performance: - AUC = 1.0: Perfect classifier - AUC = 0.5: Random guessing - AUC > 0.7: Generally acceptable

Interpretation: The probability that a randomly chosen positive example ranks higher than a randomly chosen negative example.

Equal Error Rate (EER)

The point where FPR = FNR. Lower is better.

Precision-Recall Curves

When to Use

ROC curves can be misleading with **class imbalance** (when one class is much more common).

Example: In fraud detection, 99.9% of transactions are legitimate. A model that predicts "not fraud" always achieves: - 99.9% accuracy - 100% specificity - 0% recall (catches no fraud!)

PR Curve Construction

For each threshold: 1. Compute Precision and Recall 2. Plot the point (Recall, Precision)

Key Properties

- No dependence on TN (unlike ROC)
- Sensitive to class imbalance
- Baseline is the positive class proportion

Summary Metrics

Precision @ K: Precision when retrieving top K results

Average Precision (AP): Area under the (interpolated) PR curve

mAP: Mean AP across multiple queries/classes

F-Score: Harmonic mean of precision and recall

$$F_\beta = \frac{(1+\beta^2) \cdot P \cdot R}{\beta^2 P + R}$$

- F_1: Equal weight to precision and recall
- F_2: Weights recall higher
- $F_{0.5}$: Weights precision higher

Why harmonic mean? It penalizes if either P or R is very low.

Regression Losses

Common Loss Functions

Loss	Formula	Properties
MSE	$\frac{1}{N}\sum(y-\hat{y})^2$	Sensitive to outliers; corresponds to Gaussian likelihood
MAE	$\frac{1}{N}\sum\|y-\hat{y}\|$	Robust to outliers; corresponds to Laplace likelihood

Loss	Formula	Properties
Huber	MSE for small errors, MAE for large	Best of both worlds

Quantile Loss

For predicting the q-th quantile:

$$L_q(y, \hat{y}) = \begin{cases} q \cdot (y - \hat{y}) & \text{if } y > \hat{y} \\ (1-q) \cdot (\hat{y} - y) & \text{if } y \leq \hat{y} \end{cases}$$

Asymmetric penalty: Different costs for over- vs under-prediction.

Model Calibration

A model is **well-calibrated** if its predicted probabilities match actual frequencies.

Example: Among all predictions with confidence 80%, about 80% should be correct.

Reliability Diagrams

- **x-axis**: Predicted probability (binned)
- **y-axis**: Actual proportion of positives in each bin
- **Perfect calibration**: Points fall on the diagonal

Why Calibration Matters

- For decision making, we need accurate probabilities
- Many models (especially neural networks) are overconfident
- Calibration can be fixed post-hoc (Platt scaling, isotonic regression)

Bayesian Model Selection

The Bayesian Approach

Choose the model m that maximizes posterior probability:

$$p(m|D) \propto p(D|m) \cdot p(m)$$

Marginal likelihood (evidence):

$$p(D|m) = \int p(D|\theta, m) \cdot p(\theta|m) d\theta$$

This integral automatically penalizes complexity (Occam's Razor).

Model Comparison Criteria

AIC (Akaike Information Criterion)

$$\text{AIC} = -2 \cdot \text{LL} + 2k$$

Where k = number of parameters.

Intuition: Approximates out-of-sample predictive performance.

BIC (Bayesian Information Criterion)

$$\text{BIC} = -2 \cdot \text{LL} + k \cdot \log N$$

Where N = number of observations.

Intuition: Approximates Bayesian model evidence; penalizes complexity more heavily than AIC.

AIC vs BIC

Criterion	Penalty	Selects	Best For
AIC	2k	Larger models	Prediction
BIC	k log N	Smaller models	Finding “true” model

MDL (Minimum Description Length)

Information-theoretic view:

$$\text{MDL} = L(\text{model}) + L(\text{data}|\text{model})$$

Choose the model that gives the shortest description of the data.

Frequentist Decision Theory

Risk of an Estimator

$$R(\theta, \hat{\theta}) = \mathbb{E}_{p(D|\theta)}[L(\theta, \hat{\theta}(D))]$$

The expected loss when the true parameter is θ.

Types of Risk

Bayes Risk: Average over prior on θ

$$R_B = \int R(\theta, \hat{\theta}) p(\theta) d\theta$$

Maximum Risk: Worst-case over all θ

$$R_{max} = \max_{\theta} R(\theta, \hat{\theta})$$

Empirical Risk Minimization

Population Risk: Expected loss on true distribution

$$R(f) = \mathbb{E}_{p^*(x,y)}[\ell(y, f(x))]$$

Empirical Risk: Average loss on training data

$$\hat{R}(f) = \frac{1}{N}\sum_{i=1}^{N} \ell(y_i, f(x_i))$$

The gap: Estimation error = $\hat{R}(f) - R(f)$

Structural Risk Minimization

Add complexity penalty to prevent overfitting:

$$\hat{f} = \arg\min_f \left[\hat{R}(f) + \lambda C(f)\right]$$

Statistical Learning Theory

PAC Learning

A concept is **Probably Approximately Correct (PAC) learnable** if: - With high probability (1 - δ) - We can find a hypothesis with low error (≤ ε) - Using polynomial time and data

VC Dimension

Measures the “capacity” or complexity of a hypothesis class.

Definition: Maximum number of points that can be **shattered** (perfectly classified for any labeling).

Examples: - Linear classifiers in d dimensions: VC = d + 1 - A line in 2D: VC = 3

Generalization Bounds

VC theory gives bounds like:

$$R(f) \leq \hat{R}(f) + O\left(\sqrt{\frac{\text{VC}}{N}}\right)$$

Implication: Lower VC dimension □ better generalization.

Hypothesis Testing

The Setup

- **Null hypothesis** H_0: Default assumption (e.g., "no effect")
- **Alternative hypothesis** H_1: What we want to show

Error Types

	H_0 True	H_1 True
Reject H_0	Type I Error (α)	Correct
Accept H_0	Correct	Type II Error (β)

- **Significance level** α: P(reject H_0 | H_0 true)
- **Power** 1 - β: P(reject H_0 | H_1 true)

p-value

The probability, under the null hypothesis, of observing a test statistic at least as extreme as what was observed.

Common misconception: p-value is NOT P(H_0 is true | data)!

Likelihood Ratio Test

Compare how well each hypothesis explains the data:

$$\Lambda = \frac{p(D|H_0)}{p(D|H_1)}$$

Neyman-Pearson Lemma: The likelihood ratio test is the most powerful test for a given significance level.

Summary

Concept	Key Insight
Decision Rule	Map probabilities to actions
Cost-Sensitive	Different errors have different costs
ROC Curve	Trade-off between TPR and FPR
PR Curve	Better for imbalanced classes
Calibration	Predicted probabilities should match reality
AIC/BIC	Trade-off between fit and complexity
VC Dimension	Theoretical measure of model complexity

Concept	Key Insight
Hypothesis Testing	Formal framework for statistical evidence

ewpage

Information Theory

Information theory provides mathematical tools for quantifying information, uncertainty, and the relationships between random variables. Originally developed for communication systems, it's now fundamental to machine learning.

The Big Picture

Information theory answers questions like: - How much uncertainty is in a distribution? - How different are two distributions? - How much does knowing X tell us about Y?

These concepts are central to understanding loss functions, model evaluation, and feature selection.

Entropy

Definition

Entropy measures the uncertainty or "unpredictability" of a random variable:

$$H(X) = -\sum_{x} p(x) \log p(x) = -\mathbb{E}[\log p(X)]$$

Intuition: How many bits do we need, on average, to encode samples from this distribution?

Key Properties

- **Non-negative**: $H(X) \geq 0$
- **Maximum for uniform distribution**: If all outcomes equally likely, uncertainty is maximized
- **Minimum for deterministic**: $H(X) = 0$ when outcome is certain (Dirac delta)

Examples

Fair coin: $H = -\frac{1}{2} \log \frac{1}{2} - \frac{1}{2} \log \frac{1}{2} = 1$ bit

Biased coin (p=0.9): $H = -0.9 \log 0.9 - 0.1 \log 0.1 \approx 0.47$ bits

More predictable lower entropy!

Cross-Entropy

Definition

Cross-entropy measures the average number of bits needed to encode data from distribution p using a code optimized for distribution q:

$$H(p, q) = -\sum_{x} p(x) \log q(x)$$

Intuition: How well does q approximate p?

Key Properties

- $H(p, q) \geq H(p)$ (equality when p = q)
- Cross-entropy is what we minimize in classification!

Connection to Machine Learning

When training a classifier: - **p** = true distribution (one-hot labels) - **q** = predicted distribution (softmax outputs)

Cross-entropy loss:

$$\mathcal{L} = -\sum_{c} y_c \log \hat{y}_c$$

For one-hot labels, this simplifies to: $-\log \hat{y}_{true}$

Joint and Conditional Entropy

Joint Entropy

Uncertainty about both X and Y together:

$$H(X, Y) = -\sum_{x,y} p(x, y) \log p(x, y)$$

Conditional Entropy

Remaining uncertainty about Y after observing X:

$$H(Y|X) = \sum_{x} p(x) H(Y|X = x) = -\sum_{x,y} p(x, y) \log p(y|x)$$

Chain Rule

$$H(X, Y) = H(X) + H(Y|X)$$

Intuition: Total uncertainty = uncertainty in X + remaining uncertainty in Y given X.

Perplexity

Definition

Perplexity is the exponentiated cross-entropy:

$$\text{Perplexity}(p, q) = 2^{H(p,q)}$$

Or for a sequence of N tokens:

$$\text{Perplexity} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{p(x_i)}}$$

Interpretation: The weighted average number of choices (branching factor) the model is uncertain between.

Use in Language Models

- **Lower perplexity** = better model
- Perplexity of 1 = perfect prediction
- Perplexity of V (vocabulary size) = random guessing

Example: Perplexity of 50 means the model is, on average, choosing between 50 equally likely next words.

KL Divergence

Definition

Kullback-Leibler divergence measures how different distribution q is from distribution p:

$$D_{KL}(p\|q) = \sum_{x} p(x) \log \frac{p(x)}{q(x)} = H(p, q) - H(p)$$

Intuition: Extra bits needed when using code for q to encode data from p.

Key Properties

- **Non-negative**: $D_{KL}(p\|q) \geq 0$ (Gibbs' inequality)
- **Zero iff p = q**: Perfect match
- **Asymmetric**: $D_{KL}(p\|q) \neq D_{KL}(q\|p)$ in general
- **Not a true distance** (asymmetric, no triangle inequality)

Connection to MLE

When we minimize NLL, we're minimizing:

$$\text{NLL} = -\frac{1}{N}\sum_i \log q(x_i)$$

If samples come from empirical distribution $\hat{p}$:

$$\text{NLL} = H(\hat{p}, q) = H(\hat{p}) + D_{KL}(\hat{p}\|q)$$

Since $H(\hat{p})$ is constant, **minimizing NLL = minimizing KL divergence**!

Forward vs. Reverse KL

Forward KL $D_{KL}(p\|q)$: p is the reference - Mode-covering: q tries to cover all of p's mass - Penalizes q for missing modes of p

Reverse KL $D_{KL}(q\|p)$: q is the reference - Mode-seeking: q concentrates on modes of p - Okay to miss some modes, but penalizes placing mass where p has none

---

Mutual Information

Definition

How much does knowing X tell us about Y?

$$I(X;Y) = D_{KL}(p(x,y)\|p(x)p(y))$$

Or equivalently:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

Intuition: Reduction in uncertainty about X from knowing Y.

Key Properties

- **Symmetric**: $I(X;Y) = I(Y;X)$
- **Non-negative**: $I(X;Y) \geq 0$
- **Zero iff independent**: $I(X;Y) = 0 \Leftrightarrow X \perp Y$

As Generalized Correlation

Mutual information captures **any** dependence (not just linear), making it more general than Pearson correlation.

Data Processing Inequality

If X □ Y □ Z forms a Markov chain:

$$I(X;Z) \leq I(X;Y)$$

Implication: Processing cannot increase information. You can only lose information through transformations!

Fano's Inequality

Statement

For any estimator $\hat{X}$ of X based on Y:

$$H(X|Y) \leq H(P_e) + P_e \log(|X| - 1)$$

Where $P_e = P(\hat{X} \neq X)$ is the error probability.

Implications

- **Lower bound on error**: If $H(X|Y)$ is high, error must be high
- **Feature selection**: Features with high mutual information with the target reduce classification error

---

Applications in ML

Loss Functions

Cross-entropy loss minimizes KL divergence between true and predicted distributions.

Variational Inference

Approximate intractable posterior by minimizing KL divergence.

Information Bottleneck

Find representations that maximally compress input while retaining relevant information about the output.

Data Augmentation

Spreads probability mass over larger input space, reducing overfitting.

Summary

Concept	Formula	Meaning
Entropy	$H(X) = -\mathbb{E}[\log p(X)]$	Uncertainty in X
Cross-Entropy	$H(p, q) = -\mathbb{E}_p[\log q]$	Bits to encode p using q
KL Divergence	$D_{KL}(p\|q) = H(p, q) - H(p)$	Extra bits; difference between distributions
Mutual Information	$I(X; Y) = H(X) - H(X\|Y)$	Information shared between X and Y
Perplexity	$2^{H(p,q)}$	Effective vocabulary size

ewpage

Optimization

Optimization is at the heart of machine learning — finding the parameters that minimize loss functions. This chapter covers the algorithms and techniques used to train models effectively.

The Big Picture

The optimization problem:

$$\theta^* = \arg\min_{\theta} L(\theta)$$

We want to find parameters θ that minimize some loss function L.

Challenges: - Non-convex landscapes (many local minima) - High-dimensional parameter spaces (millions of parameters) - Noisy gradients (from mini-batch sampling) - Computational constraints

---

Basic Concepts

Optima

- **Global optimum**: Best solution in the entire parameter space
- **Local optimum**: Best solution in a neighborhood (not necessarily globally best)

Optimality Conditions

For smooth functions, at a minimum: 1. **First-order condition**: Gradient is zero

$$\nabla L(\theta^*) = 0$$

2. **Second-order condition**: Hessian is positive semi-definite

$$\nabla^2 L(\theta^*) \succeq 0$$

Convexity

A function is **convex** if any local minimum is also a global minimum.

For convex functions, optimization is “easy” — gradient descent will find the global optimum.

Unfortunately: Most deep learning losses are non-convex!

Lipschitz Continuity

A function is L-Lipschitz if:

$$|f(x_1) - f(x_2)| \leq L|x_1 - x_2|$$

Interpretation: The function can't change too rapidly. This property is useful for proving convergence.

First-Order Methods

Use only gradient information (first derivatives).

Gradient Descent

The simplest optimization algorithm:

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$$

Where η is the **learning rate** or step size.

Intuition: Move in the direction of steepest descent.

Step Size Selection

Choosing η is crucial: - **Too small**: Slow convergence - **Too large**: Oscillations, divergence

Options:

1. **Constant step size**: Simple but suboptimal
2. **Line search**: Find optimal η at each step

$$\eta_t = \arg\min_{\eta} L(\theta_t - \eta \nabla L(\theta_t))$$

3. **Learning rate schedule**: Decrease η over time
    - Must satisfy Robbins-Monro conditions for convergence

Momentum

Gradient descent is slow in flat regions and oscillates in narrow valleys.

Heavy ball momentum:

$$m_t = \beta m_{t-1} + \nabla L(\theta_{t-1})$$
$$\theta_t = \theta_{t-1} - \eta m_t$$

Intuition: Accumulate velocity like a ball rolling downhill. EWMA of gradients smooths out oscillations and accelerates in consistent directions.

Typical value: β = 0.9

Nesterov Momentum

Momentum can overshoot. Nesterov adds "lookahead":

$$m_{t+1} = \beta m_t - \eta \nabla L(\theta_t + \beta m_t)$$

Idea: Compute gradient at the anticipated next position, not current position.

Second-Order Methods

Use curvature information (Hessian).

Newton's Method

Use quadratic approximation:

$$L(\theta) \approx L(\theta_t) + \nabla L^T(\theta - \theta_t) + \frac{1}{2}(\theta - \theta_t)^T H(\theta - \theta_t)$$

Optimal step:

$$\theta_{t+1} = \theta_t - H^{-1}\nabla L$$

Advantages: Faster convergence (quadratic vs. linear)

Disadvantages: - Hessian H is expensive to compute (O(d²) storage, O(d³) inversion) - Not scalable to deep learning

Quasi-Newton Methods (BFGS)

Approximate the Hessian using gradient information: - Build up Hessian approximation over iterations - **L-BFGS**: Limited memory version; uses only recent gradients

Useful for smaller models where full-batch gradients are available.

Stochastic Gradient Descent (SGD)

The Key Insight

For finite-sum problems:

$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell(y_i, f(x_i; \theta))$$

Computing the full gradient requires summing over all N examples — expensive!

SGD approximation: Use a random mini-batch B □ {1, …, N}:

$$\nabla L(\theta) \approx \frac{1}{|B|} \sum_{i \in B} \nabla \ell(y_i, f(x_i; \theta))$$

Properties

- **Unbiased**: Expected gradient equals true gradient
- **High variance**: Individual mini-batch gradients are noisy
- **Much faster**: Each step is O(|B|) instead of O(N)

Mini-batch Size Trade-offs

Batch Size	Gradient Quality	Computation	Generalization
Small	Noisy	Fast per step	Often better (regularization effect)
Large	Accurate	Slow per step, but parallelizable	May overfit

Variance Reduction

Reduce noise in SGD gradient estimates.

SVRG (Stochastic Variance Reduced Gradient)

Periodically compute full gradient; use it to correct mini-batch estimates:

$$g_t = \nabla \ell_i(\theta_t) - \nabla \ell_i(\tilde{\theta}) + \nabla L(\tilde{\theta})$$

SAGA

Maintain running estimates of gradients for each example; update incrementally.

Trade-off: Extra memory for reduced variance.

Adaptive Learning Rates

Different parameters may need different learning rates.

AdaGrad

Adapt learning rate based on historical gradient magnitudes:

$$s_t = s_{t-1} + g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t + \epsilon}} g_t$$

Effect: Parameters with large gradients get smaller learning rates.

Problem: Learning rate decreases monotonically and may become too small.

RMSProp

Use exponential moving average instead of sum:

$$s_t = \beta s_{t-1} + (1 - \beta) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t + \epsilon}} g_t$$

Prevents learning rate from vanishing.

AdaDelta

Like RMSProp, but also scales by historical update magnitudes:

$$\delta_t = \beta \delta_{t-1} + (1 - \beta)(\Delta\theta)^2$$
$$\theta_{t+1} = \theta_t - \frac{\sqrt{\delta_t + \epsilon}}{\sqrt{s_t + \epsilon}} g_t$$

Adam (Adaptive Moment Estimation)

The most popular optimizer. Combines momentum with adaptive learning rates:

First moment (mean of gradients):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Second moment (mean of squared gradients):

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) g_t^2$$

Bias correction (important for early iterations):

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$

Update:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{s}_t} + \epsilon}$$

Default values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Constrained Optimization

Lagrange Multipliers

Convert constrained to unconstrained optimization.

For equality constraint $h(\theta) = 0$:

$$\mathcal{L}(\theta, \lambda) = L(\theta) - \lambda h(\theta)$$

At optimum:

$$\nabla L = \lambda \nabla h$$

Geometric interpretation: Gradient of objective is parallel to gradient of constraint.

KKT Conditions

For inequality constraints $g(\theta) \leq 0$:

$$\mathcal{L}(\theta, \mu) = L(\theta) + \mu g(\theta)$$

Complementary slackness: - If constraint is active ($g(\theta) = 0$): $\mu > 0$ - If constraint is inactive ($g(\theta) < 0$): $\mu = 0$ - Always: $\mu \cdot g(\theta) = 0$

Proximal Gradient Descent

For composite objectives with non-smooth terms (e.g., L1 regularization):

$$L(\theta) = f(\theta) + g(\theta)$$

Where f is smooth and g is non-smooth.

1. Gradient step on smooth part
2. **Proximal operator** to handle non-smooth part:

$$\text{prox}_g(x) = \arg\min_z \left[g(z) + \frac{1}{2}\|z - x\|^2 \right]$$

Example: For L1 penalty, proximal operator is soft-thresholding.

EM Algorithm

For models with latent variables, direct MLE is difficult.

The Problem

$$\log p(Y|\theta) = \log \sum_z p(Y, z|\theta)$$

The sum inside the log is intractable.

The Solution

Iterate between:

E-step: Compute posterior over latent variables given current parameters

$$q(z) = p(z|Y, \theta^{old})$$

M-step: Maximize expected complete-data log-likelihood

$$\theta^{new} = \arg\max_\theta \mathbb{E}_{q(z)}[\log p(Y, z|\theta)]$$

Properties

- **Monotonic**: Likelihood never decreases
- **Converges**: To a local maximum
- **May get stuck**: Multiple restarts recommended

Example: GMM

- **E-step**: Compute responsibilities (soft cluster assignments)
- **M-step**: Update means, covariances, and mixing proportions

---

Simulated Annealing

For non-differentiable or discrete optimization:

1. Start with high "temperature" T
2. Propose random moves
3. Accept improvements always; accept worse moves with probability $\exp(-\Delta L/T)$
4. Gradually decrease T

Idea: High T allows escaping local minima; low T focuses on refinement.

Practical Tips

Learning Rate

- Start with a reasonable default (e.g., 0.001 for Adam)
- Use learning rate warmup for large models
- Decay learning rate during training

Initialization

- Poor initialization can prevent learning
- Xavier/Glorot: Scale by fan-in/fan-out
- He: For ReLU networks

Gradient Clipping

Prevent exploding gradients by clipping:

$$g \leftarrow \text{min}\left(1, \frac{\tau}{\|g\|}\right) g$$

Early Stopping

Monitor validation loss; stop when it starts increasing.

Summary

Method	Key Idea	When to Use
SGD	Mini-batch gradients	Large datasets
Momentum	Accumulate velocity	Faster than vanilla SGD
Adam	Adaptive + momentum	Default for deep learning
L-BFGS	Quasi-Newton	Small-medium models, full batch
Proximal	Handle non-smooth terms	L1 regularization
EM	Latent variables	Mixture models

General advice: 1. Start with Adam 2. Try SGD + momentum if Adam overfits 3. Use learning rate schedules 4. Watch for vanishing/exploding gradients

ewpage

Discriminant Analysis

Discriminant analysis covers two fundamental approaches to classification: generative and discriminative models. Understanding their differences helps you choose the right approach for your problem.

Generative vs. Discriminative Models

Discriminative Models

Model the **posterior probability** directly:

$$p(y|x, \theta)$$

Approach: Learn the decision boundary between classes.

Examples: Logistic regression, SVMs, neural networks

Advantages: - Often more accurate when we have enough data - Make fewer assumptions - More robust to model misspecification

Generative Models

Model the **joint distribution** via class-conditional densities and priors:

$$p(y = c|x, \theta) \propto p(x|y = c, \theta) \times p(y = c)$$

Components: - $p(x|y = c, \theta)$: Class-conditional density — how does data look for class c? - $p(y = c)$: Prior probability — how common is class c?

Examples: Naive Bayes, LDA, QDA, Gaussian mixture models

Advantages: - Can generate new samples - Handle missing data naturally - Work well with small datasets - Can incorporate prior knowledge

---

Gaussian Discriminant Analysis

Assume class-conditional densities are multivariate Gaussian:

$$p(x|y = c, \theta) = \mathcal{N}(x|\mu_c, \Sigma_c)$$

Quadratic Discriminant Analysis (QDA)

Each class has its own mean **and** covariance:

$$p(x|y=c) = \mathcal{N}(\mu_c, \Sigma_c)$$

Log posterior (up to constant):

$$\log p(y=c|x) = \log \pi_c - \frac{1}{2}\log|\Sigma_c| - \frac{1}{2}(x-\mu_c)^T\Sigma_c^{-1}(x-\mu_c)$$

The decision boundary is **quadratic** in x (hence the name).

Linear Discriminant Analysis (LDA)

Key assumption: All classes share the same covariance matrix.

$$\Sigma_c = \Sigma \quad \text{for all } c$$

This simplifies the log posterior to:

$$\log p(y=c|x) = \log \pi_c + x^T\Sigma^{-1}\mu_c - \frac{1}{2}\mu_c^T\Sigma^{-1}\mu_c$$

The decision boundary is **linear** in x!

Connection to logistic regression: LDA can be written in the same form, but makes stronger (Gaussian) assumptions.

Fitting GDA

Parameter estimation (usually via MLE): - $\hat{\pi}_c = N_c/N$ (class proportions) - $\hat{\mu}_c = \frac{1}{N_c}\sum_{i:y_i=c} x_i$ (class means) - $\hat{\Sigma}_c = \frac{1}{N_c}\sum_{i:y_i=c}(x_i-\hat{\mu}_c)(x_i-\hat{\mu}_c)^T$ (class covariances)

For LDA, pool covariances:

$$\hat{\Sigma} = \frac{1}{N}\sum_c \sum_{i:y_i=c}(x_i-\hat{\mu}_c)(x_i-\hat{\mu}_c)^T$$

LDA vs QDA Trade-off

Aspect	LDA	QDA
Parameters	$O(Cd+d^2)$	$O(Cd+Cd^2)$
Flexibility	Lower	Higher
Variance	Lower	Higher

Aspect	LDA	QDA
Best when	Classes have similar shapes	Classes have different shapes

Regularization: When covariance estimates are unstable, shrink towards LDA:

$$\hat{\Sigma}_c(\alpha) = \alpha\hat{\Sigma}_c + (1-\alpha)\hat{\Sigma}$$

Nearest Centroid Classifier

Classification simplifies to: assign x to the class with nearest mean (using Mahalanobis distance with Σ^{-1}).

Naive Bayes Classifiers

The Naive Independence Assumption

Assume features are **conditionally independent** given the class:

$$p(x|y=c) = \prod_{d=1}^{D} p(x_d|y=c)$$

Why "naive"? This assumption is almost never true in practice!

The Posterior

$$p(y=c|x,\theta) \propto \pi_c \prod_{d=1}^{D} p(x_d|y=c,\theta_{dc})$$

Each feature contributes independently to the log-posterior.

Feature Distributions

Choose distribution based on feature type: - **Binary features**: Bernoulli distribution - **Categorical features**: Categorical distribution - **Continuous features**: Gaussian distribution

Why Naive Bayes Works

Despite the wrong assumption: 1. **Few parameters**: Very sample-efficient 2. **Rankings often correct**: We only need relative, not absolute probabilities 3. **Errors cancel**: Overestimates and underestimates may balance out

When to Use Naive Bayes

- **Text classification**: High-dimensional, sparse features
- **Small datasets**: Fewer parameters = less overfitting
- **Fast prediction needed**: Inference is very efficient
- **As a baseline**: Simple and hard to beat in some domains

Comparing Approaches

Generative Advantages

1. **Handle missing data**: Can marginalize out missing features
2. **Semi-supervised learning**: Can use unlabeled data (via EM)
3. **Prior knowledge**: Natural way to incorporate domain knowledge
4. **Sample generation**: Can create synthetic examples

Discriminative Advantages

1. **Direct objective**: Optimize what we care about
2. **Fewer assumptions**: More robust to model misspecification
3. **Often more accurate**: With enough data
4. **Flexible**: Can model complex decision boundaries

When to Use Which

Situation	Recommendation
Small dataset	Generative (LDA, NB)
Large dataset	Discriminative (logistic, NN)
Missing features	Generative
Need probabilities	Either (both can be calibrated)
Need interpretability	LDA or logistic regression
High dimensions + sparse	Naive Bayes

Summary

Model	Assumptions	Decision Boundary	Parameters
QDA	Gaussian per class	Quadratic	$O(Cd^2)$
LDA	Gaussian, shared Σ	Linear	$O(Cd + d^2)$
Naive Bayes	Conditional independence	Can be nonlinear	$O(Cd)$

Model	Assumptions	Decision Boundary	Parameters
Logistic Regression	Linear log-odds	Linear	$O(Cd)$

Key insight: The choice between generative and discriminative is about the bias-variance trade-off. Generative models make stronger assumptions (more bias) but need less data (less variance).

ewpage

Logistic Regression

Logistic regression is the foundational discriminative model for classification. Despite its name, it's a classification algorithm (not regression) that directly models posterior class probabilities.

The Big Picture

Unlike generative models (which model how data is generated per class), logistic regression directly models:

$$p(y|x, \theta)$$

This "discriminative" approach focuses on the decision boundary rather than the full data distribution.

Binary Logistic Regression

The Model

For binary classification with $y \in \{0, 1\}$:

$$p(y = 1|x, \theta) = \sigma(w^T x + b)$$

Where σ is the **sigmoid function**:

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

The probability of class 0 is simply:

$$p(y = 0|x, \theta) = 1 - \sigma(w^T x + b) = \sigma(-(w^T x + b))$$

Alternative Notation

For $y \in \{-1, +1\}$:

$$p(y|x, \theta) = \sigma(y \cdot (w^T x + b))$$

This compact form handles both classes with one equation.

The Decision Boundary

Predict $y = 1$ if $p(y = 1|x) > 0.5$, which occurs when:

$$w^T x + b > 0$$

This is a **hyperplane** in feature space — logistic regression produces linear decision boundaries.

Geometric interpretation: - **w** is the normal vector to the decision boundary - **b** determines the offset from the origin - Distance from boundary relates to confidence

---

Maximum Likelihood Estimation

The Likelihood

For dataset $\{(x_i, y_i)\}_{i=1}^N$:

$$L(\theta) = \prod_{i=1}^{N} p(y_i|x_i, \theta)$$

Negative Log-Likelihood (Binary Cross-Entropy)

$$\text{NLL}(\theta) = -\sum_{i=1}^{N} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$

Where $\hat{y}_i = \sigma(w^T x_i + b)$.

For $y \in \{-1, +1\}$ notation:

$$\text{NLL}(\theta) = \sum_{i=1}^{N} \log(1 + \exp(-y_i(w^T x_i + b)))$$

Computing the Gradient

The gradient has a beautiful form:

$$\nabla_w \text{NLL} = \sum_{i=1}^{N} (\hat{y}_i - y_i) x_i = X^T(\hat{y} - y)$$

Intuition: The gradient is a weighted sum of input vectors, where the weights are the prediction errors.

Optimization

Good news: The NLL is **convex** (Hessian is positive semi-definite).

Methods: 1. **Gradient Descent / SGD**: Simple, works for large datasets 2. **Newton's Method**: Faster convergence for smaller problems 3. **IRLS**: Iteratively Reweighted Least Squares — Newton's method specialized for logistic regression

---

Regularization

The Overfitting Problem

MLE can overfit, especially with: - High-dimensional features - Small datasets - Linearly separable data (weights □ ∞)

L2 Regularization (Ridge)

Add Gaussian prior on weights:

$$p(w) = \mathcal{N}(0, \lambda^{-1} I)$$

Regularized objective:

$$L(w) = \text{NLL}(w) + \lambda \|w\|^2$$

Effect: Penalizes large weights, improves generalization.

L1 Regularization (Lasso)

Use Laplace prior for sparse solutions:

$$L(w) = \text{NLL}(w) + \lambda \|w\|_1$$

Effect: Some weights become exactly zero — automatic feature selection.

Practical Notes

- **Standardize features** before applying regularization (features should be on same scale)
- **Don't regularize the bias** term
- Choose λ via cross-validation

Multinomial Logistic Regression

Extending to Multiple Classes

For $y \in \{1, 2, ..., C\}$:

$$p(y = c|x, \theta) = \frac{\exp(w_c^T x + b_c)}{\sum_{j=1}^{C} \exp(w_j^T x + b_j)} = \text{softmax}(a)_c$$

Where $a_c = w_c^T x + b_c$ are the **logits**.

Overparameterization

Note: We have C sets of weights, but only C-1 are needed (one class can be the reference).

For binary case with softmax:

$$p(y = 0|x) = \frac{e^{a_0}}{e^{a_0} + e^{a_1}} = \sigma(a_0 - a_1)$$

This reduces to standard logistic regression with $w = w_0 - w_1$.

Maximum Entropy Classifier

When features depend on both x and the class c:

$$p(y = c|x, w) \propto \exp(w^T \phi(x, c))$$

This is common in NLP where features might include “word X appears AND class is Y”.

Handling Special Cases

Hierarchical Classification

When classes have taxonomy (e.g., animal □ mammal □ dog):

Label smearing: Propagate positive labels to parent categories.

Approach: Multi-label classification where an example can belong to multiple levels.

Many Classes

Hierarchical softmax: Organize classes in a tree; predict by traversing tree. - Reduces computation from O(C) to O(log C) - Put frequent classes near root

Class Imbalance

When some classes are much more common:

Resampling strategies:

$$p_c = \frac{N_c^q}{\sum_j N_j^q}$$

- q = 1: Instance-balanced (original distribution)
- q = 0: Class-balanced (equal weight per class)
- q = 0.5: Square-root sampling (compromise)

Robust Logistic Regression

Handling Outliers and Label Noise

Standard logistic regression is sensitive to mislabeled examples.

Mixture model approach:

$$p(y|x) = \pi \cdot \text{Ber}(0.5) + (1 - \pi) \cdot \text{Ber}(\sigma(w^T x + b))$$

Mix predictions with uniform noise — mislabeled points have less impact.

Bi-tempered Logistic Loss

Two modifications for robustness: 1. **Tempered cross-entropy**: Handles outliers far from boundary 2. **Tempered softmax**: Handles noise near boundary

Probit Regression

Replace sigmoid with Gaussian CDF (probit function):

$$p(y = 1|x) = \Phi(w^T x + b)$$

Similar shape to logistic but different tails — can be more robust in some cases.

Summary

Aspect	Key Points
Model	$p(y = 1 \| x) = \sigma(w^T x + b)$
Loss	Binary cross-entropy (NLL)
Optimization	Convex — guaranteed global optimum
Boundary	Linear (hyperplane)
Regularization	L2 (shrink) or L1 (sparse)
Multiclass	Softmax over C classes
Robustness	Mixture models, tempered losses

When to Use Logistic Regression

Good for: - Binary and multiclass classification - When interpretability matters (coefficients are meaningful) - As a baseline before trying complex models - When computational resources are limited

Limitations: - Linear decision boundaries - May underfit complex data - Sensitive to outliers (without modifications)

Pro tip: Start with logistic regression. If it works well, you may not need anything more complex!

ewpage

Linear Regression

Linear regression is the foundation of supervised learning for continuous outputs. Understanding it deeply gives insight into more complex models.

The Big Picture

The model:

$$p(y|x, \theta) = \mathcal{N}(y|w^T x + b, \sigma^2)$$

Translation: Given features x, the output y is normally distributed around a linear prediction, with some noise σ².

Types of Linear Regression

Type	Description
Simple	One input feature
Multiple	Many input features
Multivariate	Multiple output variables
Polynomial	Non-linear by adding x^2, x^3, etc. as features

Key insight: "Linear" refers to linearity in **parameters**, not features. Polynomial regression is still "linear regression"!

Least Squares Estimation

The Objective

Minimize the Negative Log-Likelihood:

$$\text{NLL}(w, \sigma^2) = \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \frac{N}{2} \log(2\pi\sigma^2)$$

The first term is the **Residual Sum of Squares (RSS)**.

The Normal Equations

Setting $\nabla_w \text{RSS} = 0$:

$$X^T X w = X^T y$$

Solution:

$$\hat{w} = (X^T X)^{-1} X^T y$$

Why "normal"? The residual vector $(y - Xw)$ is orthogonal (normal) to the column space of X.

Geometric Interpretation

$\hat{y} = X\hat{w}$ is the **projection** of y onto the column space of X. We find the closest point in the subspace spanned by the features.

Practical Computation

Direct matrix inversion can be numerically unstable. Better approaches: 1. **SVD**: $X = U\Sigma V^T$, then $\hat{w} = V\Sigma^{-1}U^T y$ 2. **QR decomposition**: More stable for ill-conditioned problems

Simple Linear Regression

For one feature:

$$\hat{w} = \frac{\text{Cov}(X, Y)}{\text{Var}(X)} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$$

$$\hat{b} = \bar{y} - \hat{w}\bar{x}$$

Intuition: Slope is ratio of covariance to variance. Intercept ensures line passes through $(\bar{x}, \bar{y})$.

Estimating the Noise Variance

$$\hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \frac{\text{RSS}}{N}$$

Note: This is biased! Unbiased version divides by (N - p - 1).

Goodness of Fit

Residual Analysis

Check assumptions by plotting residuals: - Should be normally distributed - Should have zero mean - Should be homoscedastic (constant variance) - Should be independent

Coefficient of Determination (R^2)

$$R^2 = 1 - \frac{\text{RSS}}{\text{TSS}} = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

Where: - **TSS** (Total Sum of Squares): Variance of y - **RSS** (Residual Sum of Squares): Unexplained variance

Interpretation: - $R^2 = 1$: Perfect fit - $R^2 = 0$: Model no better than predicting the mean - $R^2 < 0$: Model is worse than predicting the mean (possible with regularization)

RMSE (Root Mean Squared Error)

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum(y_i - \hat{y}_i)^2}$$

In same units as y — more interpretable than MSE.

Ridge Regression (L2 Regularization)

The Problem with OLS

MLE can overfit when: - Features are correlated (multicollinearity) - Number of features exceeds samples (p > N) - (X^TX) is ill-conditioned

The Ridge Solution

Add L2 penalty on weights:

$$L(w) = \text{RSS} + \lambda\|w\|^2$$

Closed-form solution:

$$\hat{w}^{\text{ridge}} = (X^TX + \lambda I)^{-1}X^Ty$$

Effect: Adding λI to the diagonal makes the matrix invertible!

Bayesian Interpretation

Ridge = MAP estimation with Gaussian prior:

$$p(w) = \mathcal{N}(0, \lambda^{-1}\sigma^2 I)$$

Connection to PCA

Ridge regression shrinks coefficients more in directions of low variance (small eigenvalues of X^TX).

Intuition: Directions with little data support get regularized more heavily.

Robust Regression

The Outlier Problem

OLS is sensitive to outliers (squared error heavily penalizes large residuals).

Solutions

1. **Student-t distribution**: Heavy tails don't penalize outliers as much
   - Fit via EM algorithm
2. **Laplace distribution**: Corresponds to L1 loss (MAE)
   - More robust than Gaussian
3. **Huber Loss**: Best of both worlds
   - L2 for small errors (smooth optimization)
   - L1 for large errors (robustness)
4. **RANSAC**: Iteratively identify and exclude outliers

Lasso Regression (L1 Regularization)

The L1 Penalty

$$L(w) = \text{RSS} + \lambda \|w\|_1 = \text{RSS} + \lambda \sum_j |w_j|$$

Sparsity!

Unlike Ridge, Lasso can set coefficients exactly to zero.

Why? Consider the Lagrangian view: - L2 constraint: $\|w\|^2 \leq B$ (sphere) - L1 constraint: $\|w\|_1 \leq B$ (diamond)

The diamond has corners on the axes. The optimal solution often hits a corner, making some weights zero.

Regularization Path

As λ decreases from ∞ to 0: - Weights "enter" the model one by one - Order of entry indicates relative importance - Use cross-validation to select optimal λ

Bayesian Interpretation

Lasso = MAP with Laplace prior:

$$p(w) \propto \exp(-\lambda |w|)$$

Elastic Net

Combining L1 and L2

$$L(w) = \text{RSS} + \lambda_1 \|w\|_1 + \lambda_2 \|w\|^2$$

Advantages

- **Sparsity** from L1
- **Grouping effect** from L2: Correlated features tend to get similar coefficients
- More stable than pure Lasso

Optimization: Coordinate Descent

The Algorithm

For Lasso and Elastic Net: 1. Initialize all weights 2. For each coordinate j: - Fix all other weights - Optimize w_j (has closed-form solution!) 3. Repeat until convergence

Why it works: Each subproblem is easy, and cycling through converges to the global optimum for convex problems.

Summary

Method	Penalty	Sparsity	Computation	Best For
OLS	None	No	Closed-form	Well-conditioned problems
Ridge	L2	No	Closed-form	Multicollinearity
Lasso	L1	Yes	Iterative	Feature selection
Elastic Net	L1 + L2	Yes	Iterative	Correlated features

Practical Tips

1. **Always visualize residuals** to check assumptions
2. **Standardize features** before regularization
3. **Use cross-validation** to choose λ
4. **Start simple** (OLS), add complexity as needed
5. **Lasso for interpretability** (sparse models)
6. **Ridge for prediction** (usually slightly better than Lasso)

ewpage

Feed-Forward Neural Networks

Neural networks are powerful function approximators that learn hierarchical representations. This chapter covers the fundamentals of deep learning.

The Big Picture

Key insight: Compose simple functions to create complex ones.

$$f(x) = f_L(f_{L-1}(...f_2(f_1(x))...))$$

Each layer transforms its input, extracting progressively more abstract features.

From Linear Models to Neural Networks

Limitations of Linear Models

Linear models: $f(x) = Wx + b$

Problem: Can only represent linear decision boundaries.

Feature Engineering

One solution: Transform features first:

$$f(x) = W\phi(x) + b$$

Where $\phi(x)$ are hand-crafted features (polynomials, interactions, etc.).

Problem: Requires domain expertise; doesn't scale.

The Neural Network Solution

Learn the features automatically!

$$f(x) = W_L \cdot \sigma(W_{L-1} \cdot \sigma(...\sigma(W_1 x + b_1)...) + b_{L-1}) + b_L$$

Each layer learns a useful transformation.

Activation Functions

Non-linear functions applied after each layer. Without them, the network would collapse to a single linear transformation.

Sigmoid

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

- Output: (0, 1)
- **Problem**: Vanishing gradients (saturates for large |x|)
- **Problem**: Not zero-centered

Tanh

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

- Output: (-1, 1)
- Zero-centered (better than sigmoid)
- Still has vanishing gradient problem

ReLU (Rectified Linear Unit)

$$\text{ReLU}(x) = \max(0, x)$$

- Output: [0, ∞)
- **Pros**: Non-saturating, computationally efficient, sparse activations
- **Cons**: “Dead ReLU” — neurons can get stuck at 0

Leaky ReLU

$$\text{LeakyReLU}(x) = \max(\alpha x, x)$$

- Small slope α (e.g., 0.01) for negative inputs
- Prevents dead neurons
- **Parametric ReLU (PReLU)**: Learn α

GELU (Gaussian Error Linear Unit)

$$\text{GELU}(x) = x \cdot \Phi(x)$$

Where Φ is the Gaussian CDF. - Smooth approximation of ReLU - Used in transformers (BERT, GPT)

Swish

$$\text{Swish}(x) = x \cdot \sigma(x)$$

- Self-gated
- Works well in deep networks

The XOR Problem

A classic example showing why we need hidden layers.

XOR truth table: | x_1 | x_2 | y | |—-|—-|—| | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 0 |

No single line can separate the classes!

With one hidden layer (2 neurons), a neural network can solve XOR by: 1. First layer creates two linear separators 2. Second layer combines them

---

Universal Approximation Theorem

Statement: A neural network with a single hidden layer of sufficient width can approximate any continuous function on a compact domain to arbitrary precision.

Implication: Neural networks are extremely powerful function approximators.

In practice: Deep (many layers) is often better than wide (many neurons per layer): - More parameter efficient - Learns hierarchical representations - Better generalization

---

Backpropagation

The algorithm that makes training deep networks possible.

The Chain Rule

For composed functions $f = f_1 \circ f_2 \circ ... \circ f_L$:

$$\frac{\partial L}{\partial \theta_l} = \frac{\partial L}{\partial z_L} \cdot \frac{\partial z_L}{\partial z_{L-1}} \cdot ... \cdot \frac{\partial z_{l+1}}{\partial z_l} \cdot \frac{\partial z_l}{\partial \theta_l}$$

Forward Pass

Compute activations layer by layer, storing intermediate values.

Backward Pass

Compute gradients layer by layer, from output to input:

$$\frac{\partial L}{\partial z_l} = \frac{\partial L}{\partial z_{l+1}} \cdot \frac{\partial z_{l+1}}{\partial z_l}$$

Automatic Differentiation

Modern frameworks (PyTorch, TensorFlow) build a computational graph and compute gradients automatically.

Forward mode: Efficient when few inputs, many outputs **Reverse mode (backprop)**: Efficient when many inputs, few outputs (typical in ML)

Example: Cross-Entropy Gradient

For softmax + cross-entropy:

$$\frac{\partial L}{\partial a_c} = p_c - y_c$$

Beautifully simple: just the prediction error!

Example: ReLU Gradient

$$\frac{\partial}{\partial x}\text{ReLU}(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Gradient flows unchanged through positive regions, is blocked through negative.

Training Challenges

Vanishing Gradients

Problem: Gradients shrink exponentially in deep networks.

Cause: Chained derivatives of saturating activations (sigmoid, tanh).

Solutions: - Use non-saturating activations (ReLU and variants) - Residual connections - Careful initialization - Batch/layer normalization

Exploding Gradients

Problem: Gradients grow exponentially.

Solutions: - Gradient clipping: $g \leftarrow \min(1, \frac{\tau}{\|g\|})g$ - Careful initialization

Mathematical Perspective

Gradient through L layers:

$$\frac{\partial L}{\partial z_1} = \prod_{l=1}^{L-1} J_l \cdot g_L$$

If eigenvalues of Jacobians are: - < 1: Gradients vanish - > 1: Gradients explode

---

Residual Connections

Key innovation (ResNet): Add skip connections.

$$z_{l+1} = z_l + f_l(z_l)$$

Benefits: - Gradients flow directly through skip connection - Learn small perturbations instead of full transformations - Enables training very deep networks (100+ layers)

Gradient flow:

$$\frac{\partial L}{\partial z_l} = \frac{\partial L}{\partial z_L} \left(1 + \frac{\partial}{\partial z_l} \sum_{i=l}^{L-1} f_i(z_i) \right)$$

The “1” term ensures gradients always flow, even if the other term vanishes.

Initialization

Poor initialization can prevent learning entirely.

The Problem

If weights are too large or too small: - Activations explode or vanish - Gradients explode or vanish

Xavier/Glorot Initialization

For linear activations:

$$w \sim \mathcal{N}\left(0, \frac{2}{n_{in} + n_{out}}\right)$$

Maintains variance of activations and gradients across layers.

He Initialization

For ReLU activations:

$$w \sim \mathcal{N}\left(0, \frac{2}{n_{in}}\right)$$

Accounts for ReLU killing half the activations.

Regularization

Early Stopping

Stop training when validation error starts increasing. - Implicit regularization - Prevents overfitting

Weight Decay (L2)

Add penalty on weight magnitudes:

$$L = L_{data} + \lambda \sum_{l} \|W_l\|^2$$

Equivalent to Gaussian prior on weights (MAP estimation).

Dropout

Randomly “drop” neurons during training with probability p.

$$h_i = \begin{cases} 0 & \text{with probability } p \\ h_i/(1-p) & \text{otherwise} \end{cases}$$

Interpretation: - Prevents co-adaptation of neurons - Approximate ensemble of subnetworks - At test time: use full network (or Monte Carlo dropout for uncertainty)

Data Augmentation

Create modified versions of training data: - Images: rotations, flips, crops, color jitter - Text: synonym replacement, back-translation

---

Layer Normalization

Normalize activations to stabilize training:

$$\hat{z} = \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\tilde{z} = \gamma \hat{z} + \beta$$

Where γ and β are learnable parameters.

Batch Norm: Normalize across batch dimension **Layer Norm**: Normalize across feature dimension (better for RNNs, Transformers)

Summary

Component	Purpose
Layers	Transform representations
Activations	Add non-linearity
Backprop	Compute gradients efficiently
Residual connections	Enable deep networks
Normalization	Stabilize training
Dropout	Prevent overfitting
Initialization	Start training successfully

Practical Recipe

1. Start with standard architecture (ResNet, etc.)
2. Use ReLU or GELU activations
3. Xavier/He initialization
4. Adam optimizer
5. Batch/Layer normalization
6. Dropout if overfitting
7. Data augmentation for images
8. Early stopping based on validation loss

ewpage

Convolutional Neural Networks

CNNs are specialized neural networks designed for processing grid-structured data, especially images. They're the foundation of modern computer vision.

The Big Picture

Problem with MLPs for images: - Different image sizes different input dimensions - Translation invariance is hard to learn - Too many parameters (e.g., 1000×1000 image = 3 million inputs!)

CNN solution: - Local connectivity (each neuron sees small region) - Weight sharing (same filter applied everywhere) - Translation equivariance built in

---

The Convolution Operation

1D Convolution

$$[w \star x]_i = \sum_{u=0}^{L-1} w_u \cdot x_{i+u}$$

Slide a **filter** (kernel) across the input and compute dot products.

2D Convolution

$$[W \star X]_{i,j} = \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} w_{u,v} \cdot x_{i+u,j+v}$$

Interpretation: Template matching. High response where input matches the filter pattern.

Key Insight: Weight Sharing

Same filter weights used at every location huge parameter reduction!

Example: 3×3 filter has 9 parameters, regardless of image size.

Convolution as Matrix Multiplication

Convolution can be expressed as multiplication by a **Toeplitz matrix**:

$$y = Cx$$

Where C has a special sparse structure with repeated weights.

This equivalence is useful for: - Understanding computational cost - Implementing on hardware

---

Convolution Variants

Valid Convolution

No padding; output shrinks: - Input: (H, W) - Filter: (f_H, f_W) - Output: $(H - f_H + 1, W - f_W + 1)$

Same (Zero) Padding

Pad input with zeros to maintain size: - Padding: $p = (f - 1)/2$ - Output same size as input

Strided Convolution

Skip positions to downsample: - Stride s: move filter by s pixels - Output size: $\lfloor (H+2p-f)/s+1 \rfloor$

Multi-Channel Convolutions

Input with Multiple Channels

For RGB images (3 channels), the filter is 3D:

$$z_{i,j} = \sum_{c} \sum_{u} \sum_{v} x_{i+u,j+v,c} \cdot w_{u,v,c}$$

Each filter produces one output channel.

Multiple Filters

To detect multiple features, use multiple filters: - Weight tensor: $(f_H, f_W, C_{in}, C_{out})$ - Each filter produces one channel of output

Output: Stack of feature maps (one per filter).

1×1 Convolution

Special case: filter size = 1×1 - Acts only across channels, not spatial - Like a per-pixel fully-connected layer - Used to change number of channels cheaply

---

Pooling Layers

Purpose

- Reduce spatial dimensions
- Achieve translation **invariance** (small shifts don't matter)
- Reduce parameters and computation

Max Pooling

Take maximum value in each window:

$$y_{i,j} = \max_{(u,v) \in \text{window}} x_{i+u,j+v}$$

Most common: 2×2 window with stride 2 (halves dimensions).

Average Pooling

Take mean instead of max.

Global Average Pooling

Average over entire spatial dimensions: - Input: (H, W, C) □ Output: $(1, 1, C)$ - Often used before final classifier

---

Dilated (Atrous) Convolution

Insert "holes" in the filter: - Dilation rate r: sample every r-th pixel - Increases **receptive field** without increasing parameters - Useful for dense prediction (segmentation)

Transposed Convolution

"Upsampling" convolution for: - Autoencoders - Generative models - Semantic segmentation

Increases spatial dimensions (opposite of regular conv).

Normalization

Batch Normalization

Normalize across the batch dimension:

$$\hat{z}_n = \frac{z_n - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\tilde{z}_n = \gamma \hat{z}_n + \beta$$

Per channel: Compute μ, σ over (N, H, W) for each channel.

Benefits: - Stabilizes training - Allows higher learning rates - Some regularization effect

Issues: - Depends on batch statistics □ problems with small batches - Different behavior at train vs. test time

Layer Normalization

Normalize across channels (and spatial dims): - Independent of batch size - Better for RNNs and Transformers

Instance Normalization

Normalize per sample, per channel: - Used in style transfer

Common Architectures

ResNet (Residual Networks)

Key innovation: Skip connections

$$y = F(x) + x$$

Residual block:

```
x → Conv → BN → ReLU → Conv → BN → (+x) → ReLU
```

Enables training 100+ layer networks.

DenseNet

Key idea: Connect each layer to all subsequent layers

$$x_l = [x_0, f_1(x_0), f_2(x_0, x_1), ...]$$

Benefits: - Feature reuse - Strong gradient flow

Drawback: Memory intensive

EfficientNet

Key insight: Scale depth, width, and resolution together - Neural Architecture Search (NAS) to find optimal scaling

---

Adversarial Examples

White-Box Attacks

Attacker has full access to model.

FGSM (Fast Gradient Sign Method):

$$x_{adv} = x + \epsilon \cdot \text{sign}(\nabla_x L)$$

Add small perturbation in gradient direction.

PGD (Projected Gradient Descent): Iterative version of FGSM; stronger attack.

Black-Box Attacks

No access to model internals: - Query-based attacks - Transfer attacks (adversarial examples transfer across models)

Defenses

- Adversarial training
- Input preprocessing
- Certified defenses (provable robustness)

Summary

Component	Purpose
Convolution	Local feature detection with weight sharing
Pooling	Downsample, add invariance
Stride	Alternative to pooling for downsampling
Padding	Control output size
1×1 Conv	Channel mixing
Skip connections	Enable deep networks
Normalization	Stabilize training

Why CNNs Work for Images

1. **Local structure**: Nearby pixels are related
2. **Translation equivariance**: Features can appear anywhere
3. **Hierarchical composition**: Simple features complex objects
4. **Parameter efficiency**: Weight sharing dramatically reduces parameters

Practical Tips

1. Use pre-trained models when possible (transfer learning)
2. Start with proven architectures (ResNet, EfficientNet)
3. Data augmentation is crucial
4. Batch normalization helps training
5. Global average pooling instead of flattening before classifier

ewpage

Recurrent Neural Networks and Transformers

RNNs process sequential data by maintaining a hidden state that carries information across time steps. Transformers, which use attention mechanisms, have largely replaced RNNs for many tasks.

The Big Picture

Sequential data is everywhere: text, speech, time series, video.

The challenge: Variable-length inputs with temporal dependencies.

RNN solution: Maintain a memory (hidden state) that updates as new inputs arrive.

Transformer solution: Use attention to relate all positions directly.

Core RNN Architecture

The Basic Update

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

Components: - x_t: Input at time t - h_{t-1}: Previous hidden state (the "memory") - W_{xh}: Input-to-hidden weights - W_{hh}: Hidden-to-hidden weights (recurrent connection) - ϕ: Non-linearity (usually tanh)

Key insight: Same weights used at every time step (weight sharing through time).

Types of Sequence Tasks

Seq2Vec (Many-to-One)

Variable-length input □ Fixed output

Examples: Sentiment analysis, document classification

Approach: Use final hidden state (or aggregate all states) as representation.

Vec2Seq (One-to-Many)

Fixed input □ Variable-length output

Examples: Image captioning, music generation

Approach: Condition on input vector, generate sequence autoregressively.

Seq2Seq (Many-to-Many)

Variable input □ Variable output

Examples: Machine translation, summarization

Approach: Encoder-decoder architecture.

Bidirectional RNNs

Process sequence in both directions:

Forward: $\vec{h}_t = f(x_t, \vec{h}_{t-1})$ **Backward**: $\overleftarrow{h}_t = f(x_t, \overleftarrow{h}_{t+1})$

Final state: Concatenate both: $h_t = [\vec{h}_t; \overleftarrow{h}_t]$

Benefit: Each position has access to both past and future context.

Limitation: Can't be used for autoregressive generation (need future that doesn't exist yet).

The Vanishing/Exploding Gradient Problem

The Problem

Gradient through L time steps:

$$\frac{\partial L}{\partial h_0} = \prod_{t=1}^{L} \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial L}{\partial h_L}$$

If $\|W_{hh}\| < 1$: Gradients vanish exponentially If $\|W_{hh}\| > 1$: Gradients explode exponentially

Exploding Gradient Solution

Gradient clipping:

$$g \leftarrow \text{min}\left(1, \frac{\tau}{\|g\|}\right) g$$

Vanishing Gradient Solutions

Use architectures with **additive updates** instead of multiplicative: - LSTM - GRU - Skip connections

---

LSTM (Long Short-Term Memory)

The Key Innovation

Separate **cell state** C_t that flows through time with minimal transformation.

Gates

Three gates control information flow:

Forget Gate: What to discard from cell state

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

Input Gate: What new information to add

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

Output Gate: What to output from cell state

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

Update Equations

Candidate cell state:

$$\tilde{C_t} = \text{tanh}(W_c[h_{t-1}, x_t] + b_c)$$

Cell state update (additive!):

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C_t}$$

Hidden state output:

$$h_t = o_t \odot \text{tanh}(C_t)$$

Why LSTM Works

The cell state acts like a “conveyor belt” — gradients can flow through unchanged if the forget gate is open.

GRU (Gated Recurrent Unit)

Simplified version of LSTM with fewer parameters.

Two gates: - **Update gate** z_t: How much to update hidden state - **Reset gate** r_t: How much past state to forget

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Trade-off: Fewer parameters, competitive performance.

Backpropagation Through Time (BPTT)

The Algorithm

1. Unroll network through time
2. Forward pass: Compute all hidden states
3. Backward pass: Compute gradients through the unrolled graph
4. Sum gradients for shared weights across time

Truncated BPTT

For long sequences: - Don't backprop through entire sequence - Truncate to manageable length (e.g., 100 steps) - Trade-off: Can't learn very long-range dependencies

---

Decoding Strategies

Greedy Decoding

At each step, pick the most likely token:

$$y_t = \arg\max_y P(y|y_{<t}, x)$$

Problem: Locally optimal choices may not be globally optimal.

Beam Search

Keep top-K candidates at each step: 1. Expand each candidate in all possible ways 2. Keep the K highest-probability sequences 3. Continue until all sequences end

Benefit: Better than greedy; balances quality and computation.

Sampling

For generation diversity:

Top-K sampling: Sample from top K tokens only

Top-P (nucleus) sampling: Sample from smallest set with cumulative probability > p

Temperature: Scale logits before softmax - Low T: More deterministic - High T: More random

---

Attention Mechanism

The Problem with Basic RNNs

All information must flow through the bottleneck of the hidden state.

For long sequences, early information gets “washed out.”

The Attention Solution

Allow the decoder to “look at” all encoder states:

$$\text{Attention}(q, (k_1, v_1), ..., (k_m, v_m)) = \sum_{i=1}^{m} \alpha_i \cdot v_i$$

Where attention weights α_i depend on similarity between query q and keys k_i.

Scaled Dot-Product Attention

$$\alpha_i = \text{softmax}\left(\frac{q^T k_i}{\sqrt{d}}\right)$$

Scaling by $\sqrt{d}$: Prevents softmax saturation when dimensions are large.

Seq2Seq with Attention

Instead of using only the final encoder state: - Query: Current decoder hidden state - Keys & Values: All encoder hidden states

Context at each decoding step:

$$c_t = \sum_i \alpha_i(h_t^{dec}, h_i^{enc}) \cdot h_i^{enc}$$

Transformers

The Revolution

Key insight: Attention is all you need — no recurrence required!

Benefits: - Parallelizable (no sequential dependency) - Direct connections between all positions - Scales to very long sequences

Self-Attention

Each position attends to all positions (including itself):

$$y_i = \text{Attention}(x_i, (x_1, x_1), (x_2, x_2), ..., (x_n, x_n))$$

Query, Key, Value: All derived from same input via learned projections.

Multi-Head Attention

Run multiple attention operations in parallel:

$$h_i = \text{Attention}(W_i^Q x, W_i^K x, W_i^V x)$$

$$\text{output} = \text{Concat}(h_1, ..., h_H) W^O$$

Benefit: Capture different types of relationships.

Positional Encoding

Attention is permutation-invariant — it doesn't know position!

Solution: Add position information to inputs:

$$x_{pos} = x + \text{PE}(pos)$$

Sinusoidal encoding (original Transformer): - Different frequencies for different dimensions - Can generalize to unseen lengths

Learned embeddings (common in practice).

Transformer Architecture

Encoder block: 1. Multi-head self-attention + residual + LayerNorm 2. Feed-forward network + residual + LayerNorm

Decoder block: 1. Masked self-attention (can't see future) 2. Cross-attention to encoder 3. Feed-forward network

---

Pre-trained Language Models

ELMo

Concatenate forward and backward LSTM representations.

BERT (Bidirectional Encoder)

Pre-training tasks: - Masked Language Modeling (MLM): Predict masked tokens - Next Sentence Prediction

Fine-tuning: Add task-specific head.

GPT (Generative Pre-Training)

Architecture: Decoder-only transformer (causal masking)

Pre-training: Autoregressive language modeling

Key insight: Scale up model and data emergent capabilities.

T5 (Text-to-Text Transfer Transformer)

Unifying framework: Every task as text-to-text - Classification: “classify: text label” - Translation: “translate: source target”

Summary

Architecture	Key Feature	Best For
Basic RNN	Recurrent hidden state	Short sequences
LSTM/GRU	Gates + additive updates	Medium sequences
Bidirectional	Both directions	When future is available
Attention	Direct access to all positions	Long-range dependencies
Transformer	Self-attention + parallelism	Everything (modern default)

When to Use What

- **RNN/LSTM**: Small data, limited compute, streaming data
- **Transformer**: Large data, sufficient compute, best performance
- **Pre-trained models**: Almost always start here and fine-tune!

ewpage

Exemplar-Based Methods

Exemplar methods (also called instance-based or memory-based) keep training data around and use it directly for prediction. The classic example is K-Nearest Neighbors (KNN).

The Big Picture

Parametric models: Learn parameters θ, discard training data at test time. - Parameters: Fixed, doesn't grow with data

Non-parametric models: Keep training data, use it directly. - Model complexity grows with data size - Can adapt to arbitrary complexity

---

Instance-Based Learning

The Approach

1. Store training examples
2. At test time, find similar training examples
3. Predict based on their labels

Key ingredient: A good **distance/similarity measure**.

K-Nearest Neighbors (KNN)

Classification

Find K closest training points; vote on the label:

$$p(y = c|x, D) = \frac{1}{K} \sum_{i \in N_K(x)} I\{y_i = c\}$$

Hyperparameter K: - K = 1: Highly flexible, noisy - K = N: Predicts majority class always - Typical: K = 5-10, or tune via cross-validation

Regression

Average the labels of K nearest neighbors:

$$\hat{y} = \frac{1}{K} \sum_{i \in N_K(x)} y_i$$

Distance Metrics

Euclidean distance:

$$d(x, x') = \|x - x'\|_2 = \sqrt{\sum_j (x_j - x'_j)^2}$$

Mahalanobis distance (accounts for correlations):

$$d_M(x, x') = \sqrt{(x - x')^T M (x - x')}$$

Where M is a positive definite matrix (often $M = \Sigma^{-1}$).

If M = I: Reduces to Euclidean distance.

The Curse of Dimensionality

The Problem

In high dimensions, distances become meaningless: - All points become approximately equidistant - Local neighborhoods become empty - Need exponentially more data to fill space

Example

Consider the fraction of volume within 10% of the edges: - 1D: 20% - 10D: 89% - 100D: 99.99999…%

Almost all points are near the boundary!

Solutions

1. **Dimensionality reduction** (PCA, autoencoders)
2. **Feature selection**
3. **Metric learning** (learn a better distance)

Computational Efficiency

Naive Approach

For each query, compute distance to all N training points. - Time: O(Nd) per query - Infeasible for large datasets

Approximate Nearest Neighbors

KD-Trees: - Binary tree that partitions space - O(log N) for low dimensions - Degrades in high dimensions

Locality-Sensitive Hashing (LSH): - Hash similar items to same bucket - Sublinear query time - Approximate, not exact

---

Open Set Recognition

Standard classification: All test classes seen during training.

Open set: New, unseen classes may appear at test time.

KNN advantage: Can handle novel classes naturally by looking at nearest neighbors.

Applications: - Person re-identification - Anomaly detection - Few-shot learning

---

Learning Distance Metrics

Motivation

Euclidean distance may not reflect true similarity.

Goal: Learn a distance metric M that captures task-relevant similarity.

Large Margin Nearest Neighbors (LMNN)

Learn M such that: 1. Points with same label are close 2. Points with different labels are far (by margin m)

Constraint: $M = W^T W$ ensures positive definiteness.

Deep Metric Learning

The Idea

Learn an embedding function $f(x; \theta)$ such that: - Similar examples are close in embedding space - Dissimilar examples are far

Siamese Networks

Two copies of same network process two inputs.

Contrastive Loss:

$$L = I\{y_i = y_j\} \cdot d(x_i, x_j)^2 + I\{y_i \neq y_j\} \cdot [m - d(x_i, x_j)]_+^2$$

- Same class: Minimize distance
- Different class: Push apart (up to margin m)

Triplet Loss

Use triplets: (anchor, positive, negative)

$$L = [m + d(a, p) - d(a, n)]_+$$

Goal: Anchor should be closer to positive than negative by margin m.

Hard Negative Mining

Random negatives are too easy (already far from anchor).

Solution: Sample hard negatives that are close to anchor but from different class.

Connection to Cosine Similarity

For normalized embeddings:

$$\|e_1 - e_2\|^2 = 2(1 - \cos\theta)$$

Euclidean distance and cosine similarity are equivalent for unit vectors.

Kernel Density Estimation

The Idea

Estimate the probability density by placing kernels at each data point:

$$\hat{p}(x) = \frac{1}{N} \sum_{n=1}^{N} K_h(x - x_n)$$

Gaussian kernel:

$$K_h(x) = \frac{1}{h\sqrt{2\pi}} \exp\left(-\frac{x^2}{2h^2}\right)$$

Bandwidth h

Controls smoothness: - Small h: Spiky, overfitting - Large h: Over-smooth, underfitting

Choose via cross-validation.

Connection to GMM

KDE is like a GMM where: - Each point is its own cluster center - All clusters have same (spherical) covariance - No mixing proportions to learn

KDE for Classification

Use Bayes rule with class-conditional densities:

$$p(y = c|x) \propto \pi_c \cdot \hat{p}(x|y = c)$$

KDE vs KNN

Connection: Both use local neighborhoods.

- **KDE**: Fixed bandwidth, variable number of neighbors
- **KNN**: Variable bandwidth (grows until K neighbors), fixed number of neighbors

Dual view: KNN adapts to local density automatically.

Summary

Method	Key Idea	Pros	Cons
KNN	Vote of K nearest neighbors	Simple, no training	Slow at test time
Metric Learning	Learn task-specific distance	Better than Euclidean	Requires labeled pairs
Deep Metric	Embed + distance	Handles high dimensions	Needs lots of data
KDE	Kernels at each point	Density estimation	Curse of dimensionality

When to Use Exemplar Methods

Good for: - Few training examples per class (few-shot learning) - Classes change over time (no retraining needed) - Interpretable predictions (“similar to example X”) - Baseline before trying complex methods

Challenges: - High-dimensional data (curse of dimensionality) - Large training sets (computational cost) - Need good distance metric

ewpage

Decision Trees and Ensembles

Decision trees are intuitive models that partition the feature space into regions. While single trees are prone to overfitting, ensemble methods (Random Forests, Boosting) combine many trees for powerful, robust predictions.

The Big Picture

Trees: Recursively partition space with simple rules. - Highly interpretable - But high variance (unstable)

Ensembles: Combine many trees. - Random Forests: Reduce variance through averaging - Boosting: Reduce bias through sequential correction

---

Decision Tree Structure

The Model

$$f(x) = \sum_{j=1}^{J} w_j \cdot I\{x \in R_j\}$$

Where: - R_j are disjoint regions (leaves) - w_j is the prediction for region j - I{} is indicator function

Building a Tree

At each node i: 1. Select a feature d_i 2. Select a threshold t_i 3. Split: left if $x_{d_i} \leq t_i$, right otherwise

Result: Axis-parallel partitions of the feature space.

Leaf Predictions

Regression: Average of training labels in region

$$w_j = \frac{\sum_{n:x_n \in R_j} y_n}{\sum_{n:x_n \in R_j} 1}$$

Classification: Majority vote or probability distribution

Finding Optimal Splits

The Greedy Algorithm

Tree optimization is NP-hard. We use a greedy approach:

At each node, find the best split by minimizing:

$$L = \frac{|D_L|}{|D|}C_L + \frac{|D_R|}{|D|}C_R$$

Where C is the cost (impurity) and D is the data reaching that node.

Splitting Criteria

For Regression (MSE):

$$C = \frac{1}{|D|}\sum_{i \in D}(y_i - \bar{y})^2$$

For Classification:

Gini Index:

$$C = \sum_{c=1}^{C} \hat{p}_c(1 - \hat{p}_c)$$

Probability of misclassifying a randomly chosen element.

Cross-Entropy:

$$C = -\sum_{c=1}^{C} \hat{p}_c \log \hat{p}_c$$

Information-theoretic measure of impurity.

Why Binary Splits?

- More splits = more data fragmentation
- Binary splits are sufficient (can always split further)
- Simpler to optimize

Regularization (Preventing Overfitting)

Option 1: Early Stopping

Stop growing when: - Maximum depth reached - Minimum samples per leaf - Improvement below threshold

Problem: May stop too early (miss good splits downstream).

Option 2: Grow and Prune

1. Grow full tree (until pure leaves or minimum samples)
2. Prune back using cost-complexity criterion:

$$C_\alpha(T) = \sum_{j=1}^{|T|} N_j \cdot C_j + \alpha|T|$$

Where $|T|$ is number of leaves and α is complexity penalty.

Use cross-validation to select optimal α.

Handling Missing Features

Categorical: Treat “missing” as new category.

Continuous: Use surrogate splits — find alternative splits that best mimic the primary split.

Pros and Cons of Trees

Advantages

- **Interpretable**: Easy to visualize and explain
- **Minimal preprocessing**: Handles mixed types, no normalization needed
- **Fast**: Prediction is O(log nodes)
- **Robust to outliers**: Splits don’t depend on magnitudes

Disadvantages

- **High variance**: Small data changes □ different tree
- **Axis-aligned only**: Can’t capture diagonal boundaries efficiently
- **Prone to overfitting**: Without regularization

Ensemble Learning

The Core Idea

Combine multiple models to reduce errors.

Regression: Average predictions

$$\hat{y} = \frac{1}{M}\sum_{m=1}^{M} f_m(x)$$

Classification: Majority vote or average probabilities

Why Ensembles Work

For M independent classifiers each with accuracy p > 0.5:

$$P(\text{majority correct}) = \sum_{k>M/2} \binom{M}{k} p^k (1-p)^{M-k}$$

As M □ ∞, this probability □ 1!

Key requirement: Classifiers must be diverse (uncorrelated errors).

Stacking

Learn weights for combining models:

$$\hat{y} = \sum_{m} w_m f_m(x)$$

Train weights on held-out data to avoid overfitting.

Bagging (Bootstrap Aggregating)

Algorithm

1. Create B bootstrap samples (sample with replacement)
2. Train a tree on each bootstrap sample
3. Average predictions (regression) or vote (classification)

Key Properties

- Each bootstrap sample contains ~63% of unique points:

$$P(\text{included}) = 1 - (1 - 1/N)^N \approx 1 - 1/e \approx 0.632$$

- **OOB Error**: Evaluate each tree on its out-of-bag samples (free cross-validation!)

Variance Reduction

$$\text{Var}(\bar{f}) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

Where ρ is correlation between trees.

- More trees (larger B) □ second term vanishes
- Less correlation (smaller ρ) □ first term shrinks

Random Forests

Beyond Bagging

Bagging helps, but trees from similar data are correlated.

Random Forest innovation: Add randomness to splits.

At each split: 1. Randomly select m features (typically $m = \sqrt{p}$ for classification, $m = p/3$ for regression) 2. Find best split among only those m features

Why It Works

- Forces trees to use different features
- Reduces correlation between trees
- Combined with bagging □ powerful ensemble

Extra Trees

Even more randomness: - Random feature subset (like RF) - Random threshold selection (not optimized) - Faster training, often similar performance

---

Boosting

The Key Idea

Sequentially fit weak learners, each focusing on previous mistakes.

$$F_m(x) = F_{m-1}(x) + \beta_m f_m(x)$$

Boosting reduces bias (unlike bagging which reduces variance).

AdaBoost

For classification with exponential loss:

$$L(y, F) = \exp(-yF(x)), \quad y \in \{-1, +1\}$$

Algorithm: 1. Initialize equal weights on examples 2. For each round: - Train weak learner on weighted examples - Increase weights on misclassified examples - Compute learner weight based on accuracy

Gradient Boosting

Generalize to any differentiable loss:

1. Initialize: $F_0 = \arg\min_\gamma \sum L(y_i, \gamma)$
2. For m = 1 to M:
   - Compute pseudo-residuals: $r_i = -\frac{\partial L(y_i, F)}{\partial F}|_{F_{m-1}}$
   - Fit weak learner to pseudo-residuals
   - Line search for step size
   - Update: $F_m = F_{m-1} + \eta \cdot f_m$

For MSE loss: Pseudo-residuals = actual residuals!

Regularization via shrinkage: Small learning rate η (0.01-0.1) + more trees.

Stochastic Gradient Boosting

Subsample data at each round: - Faster training - Better generalization (regularization effect)

XGBoost

Innovations

1. **Regularized objective**:

$$L = \sum L(y_i, F(x_i)) + \gamma J + \frac{\lambda}{2} \sum w_j^2$$

2. **Second-order approximation** (use Hessian):

$$L \approx \sum [g_i F(x_i) + \frac{1}{2} h_i F(x_i)^2] + \text{regularization}$$

3. **Optimal leaf weights**:

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

Where $G_j = \sum_{i \in j} g_i$ and $H_j = \sum_{i \in j} h_i$.

4. **Split gain**:

$$\text{Gain} = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \gamma$$

γ acts as regularization — won't split unless gain exceeds γ.

Feature Importance

Mean Decrease in Impurity

Sum the impurity decrease at all splits using feature k:

$$R_k = \sum_{\text{nodes using } k} \Delta\text{impurity}$$

Average across all trees.

Caveat: Biased toward high-cardinality features.

Permutation Importance

1. Compute baseline accuracy
2. For each feature k:
   - Permute (shuffle) feature k's values
   - Compute accuracy drop
3. Importance = accuracy drop

More reliable but slower.

Partial Dependence Plots

Visualize effect of feature on prediction:

$$\bar{f}_k(x_k) = \frac{1}{N} \sum_{i=1}^{N} f(x_k, x_{i,-k})$$

Average over all other features.

Summary

Method	Reduces	Training	Key Hyperparameters
Single Tree	—	Fast	max_depth, min_samples
Bagging	Variance	Parallel	n_estimators
Random Forest	Variance	Parallel	n_estimators, max_features
Boosting	Bias	Sequential	n_estimators, learning_rate, max_depth

Practical Recommendations

1. **Start with Random Forest**: Works well with minimal tuning
2. **Try XGBoost/LightGBM**: Often best for tabular data
3. **Tune carefully**: Learning rate and n_estimators together
4. **Early stopping**: Monitor validation error
5. **Feature importance**: Helps interpretability

ewpage

Self-Supervised and Semi-Supervised Learning

These techniques leverage unlabeled data to improve learning. In a world where labels are expensive but data is abundant, these methods are increasingly important.

The Big Picture

The label bottleneck: Labeling data is expensive and time-consuming.

The opportunity: Vast amounts of unlabeled data are available.

The goal: Learn useful representations from unlabeled data that transfer to downstream tasks.

Data Augmentation

The Idea

Create modified versions of training examples that preserve the label.

Effect: Expands training set, improves robustness.

Common Augmentations

Images: - Rotation, flipping, cropping - Color jitter, brightness changes - Random erasing, cutout

Text: - Synonym replacement - Back-translation - Random insertion/deletion

Audio: - Pitch shifting - Time stretching - Adding noise

Theoretical View: Vicinal Risk Minimization

Instead of minimizing risk at exact training points, minimize in a **vicinity** around them:

$$R = \int L(y, f(x'))p(x'|x)dx'$$

Where p(x'|x) is the augmentation distribution.

Transfer Learning

The Problem

Task A has lots of data; Task B has little data.

The Solution

1. **Pretrain** on large source dataset (Task A)
2. **Fine-tune** on small target dataset (Task B)

Fine-tuning Strategies

Feature extraction: Freeze pretrained layers, train only new head. - Best when: Very small target data, similar domains

Full fine-tuning: Update all parameters. - Best when: More target data, different domains

Gradual unfreezing: Start from top layers, progressively unfreeze. - Often best practice

Parameter-Efficient Fine-tuning

For large models, updating all parameters is expensive.

Adapters: Small bottleneck layers inserted between frozen transformer blocks.

LoRA: Low-rank updates to weight matrices.

Prompt tuning: Learn soft prompts while keeping model frozen.

Self-Supervised Learning

The Core Idea

Create supervisory signals from the data itself — no human labels needed.

Pretext Tasks

Reconstruction: Predict masked or corrupted parts - Autoencoders - Masked language modeling (BERT) - Masked image modeling (MAE)

Contrastive: Learn to distinguish similar from dissimilar - Positive pairs: Same image, different augmentations - Negative pairs: Different images

Predictive: Predict properties of the data - Rotation prediction (images) - Next word prediction (language)

Contrastive Learning

The Framework

1. Create two views of same example (via augmentation)
2. Push their representations together
3. Push representations of different examples apart

SimCLR (Simple Contrastive Learning)

Pretraining: 1. Take image x 2. Apply two augmentations: x_1, x_2 3. Encode both: $z_1 = g(f(x_1)), z_2 = g(f(x_2))$ 4. Contrastive loss (NT-Xent):

$$L = -\log \frac{\exp(\text{sim}(z_1, z_2)/\tau)}{\sum_{k \neq i} \exp(\text{sim}(z_1, z_k)/\tau)}$$

Fine-tuning: Discard projection head g, fine-tune encoder f.

Key Insights

- **Projection head** is crucial during pretraining but discarded after
- **Large batch sizes** help (more negatives)
- **Strong augmentation** is important

Challenges

- Need many negative examples
- Batch size dependence
- Risk of feature collapse

Non-Contrastive Methods

BYOL (Bootstrap Your Own Latent)

No negative examples needed!

Architecture: - **Online network** (student): Updated by gradient descent - **Target network** (teacher): Exponential moving average of online

Loss: Predict target representation from online prediction.

Why no collapse? Asymmetric architecture + momentum updates prevent trivial solutions.

Masked Autoencoders (MAE)

Inspired by BERT's success: 1. Mask large portion of image (75%!) 2. Encode visible patches 3. Decode to reconstruct masked patches 4. For downstream: Use only encoder

Why mask so much? Forces learning high-level features, not just copying.

Semi-Supervised Learning

Use both labeled and unlabeled data together.

Self-Training

1. Train on labeled data
2. Predict on unlabeled data (pseudo-labels)
3. Add confident predictions to training set
4. Repeat

Connection to EM: Pseudo-labels are the E-step!

Noise Student Training

Self-training + noise: 1. Train teacher on labeled data 2. Generate pseudo-labels for unlabeled data 3. Train student on all data with noise (dropout, augmentation) 4. Student becomes new teacher; repeat

Key insight: Noise makes student more robust than teacher.

Consistency Regularization

Model predictions should be consistent under small input changes:

$$L = L_{supervised} + \lambda \cdot d(f(x), f(\text{aug}(x)))$$

FixMatch: Combine pseudo-labeling with consistency.

Label Propagation

Graph-Based Approach

1. Build graph: Nodes = data points, edges = similarity
2. Propagate labels from labeled to unlabeled nodes
3. Use resulting labels for training

Algorithm

- T: Transition matrix (normalized edge weights)
- Y: Label matrix (N × C)
- Iterate: $Y = TY$ until convergence
- Use propagated labels for supervised learning

Assumptions

- Similar points should have similar labels
- Cluster structure in data reflects label structure

Generative Self-Supervised Learning

Variational Autoencoders (VAE)

Generative model: 1. Sample latent: $z \sim p(z)$ 2. Generate: $x \sim p(x|z)$

Training: Maximize ELBO (Evidence Lower Bound)

$$\log p(x) \geq \mathbb{E}_{q(z|x)}[\log p(x|z)] - D_{KL}(q(z|x) \| p(z))$$

Use for SSL: Learn representations z for downstream tasks.

GANs

Generator: Maps noise to data. **Discriminator**: Distinguishes real from fake.

Semi-supervised extension: Discriminator predicts K classes + “fake”.

Active Learning

The Idea

If we must label data, label the most informative examples.

Strategies

Uncertainty sampling: Label examples where model is least confident.

$$x^* = \arg\max_x H[p(y|x)]$$

BALD (Bayesian Active Learning by Disagreement): Label where model’s predictions are most diverse (across ensemble/dropout samples).

Query by committee: Multiple models vote; label where they disagree.

Few-Shot Learning

The Challenge

Learn to classify new classes from very few examples (1-5 per class).

Meta-Learning Approach

Train model to learn quickly: - **Training**: Many "episodes" with different class subsets - **Testing**: New classes, few examples each

Metric Learning Approach

Learn embedding space where similarity = class membership. - Prototypical networks: Classify by nearest class prototype - Matching networks: Weighted nearest neighbor

---

Weak Supervision

When Labels Are Imperfect

- Noisy labels (some wrong)
- Soft labels (probability distributions)
- Aggregate from multiple labelers

Label Smoothing

Instead of hard labels [0, 1, 0]:

$$y_{smooth} = (1 - \epsilon) \cdot y + \epsilon / K$$

Prevents overconfidence, improves calibration.

Summary

Method	Uses Unlabeled Data	Key Idea
Data Augmentation	No (extends labeled)	Transform while preserving label
Transfer Learning	Pre-training stage	Leverage large datasets
Contrastive Learning	Yes	Pull similar, push dissimilar

Method	Uses Unlabeled Data	Key Idea
Non-Contrastive	Yes	Predict across views (no negatives)
Semi-Supervised	Yes (with some labels)	Pseudo-labels + consistency
Active Learning	Selects what to label	Query most informative

Practical Recommendations

1. **Always use data augmentation** — almost free improvement
2. **Start with pretrained models** — rarely worth training from scratch
3. **Try self-training** for semi-supervised (simple and effective)
4. **Large-scale pretraining** for best representations (if compute allows)
5. **Match pretraining and downstream domains** for best transfer

ewpage

Recommendation Systems

Recommendation systems predict user preferences for items (movies, products, songs, etc.). They power personalization across the internet — from Netflix to Amazon to Spotify.

The Big Picture

The problem: Users interact with a tiny fraction of available items. Can we predict what they'd like?

Key challenge: The user-item matrix is extremely sparse (>99% missing).

Goal: Fill in the missing entries (predict ratings) or rank items for each user.

Types of Feedback

Explicit Feedback

Users directly express preferences: - Star ratings (1-5) - Thumbs up/down - Reviews

Pros: Clear signal about preferences. **Cons**: Sparse (users rarely rate), not missing at random (users rate things they care about).

Implicit Feedback

Inferred from user behavior: - Clicks, views, purchases - Time spent - Add to cart

Pros: Abundant, always available. **Cons**: Noisy, positive-only (absence doesn't mean dislike).

Collaborative Filtering

The Core Idea

Users "collaborate" to help recommend items: - Users with similar preferences in the past will have similar preferences in the future - Items liked by similar users are likely to be liked by the target user

User-Based CF

For target user u and item i:

$$\hat{y}_{ui} = \frac{\sum_{u'} \text{sim}(u, u') \cdot y_{u'i}}{\sum_{u'} \text{sim}(u, u')}$$

Steps: 1. Find users similar to u (based on rating patterns) 2. Aggregate their ratings for item i

Item-Based CF

For target user u and item i:

$$\hat{y}_{ui} = \frac{\sum_{i'} \text{sim}(i, i') \cdot y_{ui'}}{\sum_{i'} \text{sim}(i, i')}$$

Steps: 1. Find items similar to i (based on who rated them) 2. Aggregate user u's ratings for those items

In practice: Item-based often preferred (item similarities more stable than user similarities).

Challenges

- **Sparsity**: Few common ratings for similarity calculation
- **Scalability**: Computing all pairwise similarities is expensive
- **Cold start**: No history for new users/items

Matrix Factorization

The Idea

The rating matrix R can be approximated as a product of low-rank matrices:

$$R \approx U \cdot V^T$$

Where: - U is user matrix (N × K): Each row is a user's "embedding" - V is item matrix (M × K): Each row is an item's "embedding" - K « N, M (typically K = 10-200)

Interpretation

Each dimension captures a latent "factor": - Movie factors might capture: Action content, Romance level, Production year… - User factors capture: Preference for action, romance, etc.

Prediction: $\hat{y}_{ui} = u_u^T v_i$ (dot product of embeddings)

Training

Minimize squared error on observed ratings:

$$L = \sum_{(u,i) \in \text{observed}} (y_{ui} - u_u^T v_i)^2 + \lambda(\|U\|^2 + \|V\|^2)$$

Note: Can't use SVD directly (missing values). Use: - **Alternating Least Squares (ALS)**: Fix U, solve for V; fix V, solve for U - **SGD**: Stochastic gradient descent on observed entries

Adding Biases

Users and items have inherent tendencies:

$$\hat{y}_{ui} = \mu + b_u + c_i + u_u^T v_i$$

Where: - μ: Global average rating - b_u: User bias (some users rate higher on average) - c_i: Item bias (some items are generally liked more)

Probabilistic Matrix Factorization

Bayesian Approach

Model ratings as:

$$p(y_{ui}|u_u, v_i) = \mathcal{N}(\mu + b_u + c_i + u_u^T v_i, \sigma^2)$$

Add priors on embeddings:

$$u_u \sim \mathcal{N}(0, \sigma_u^2 I)$$
$$v_i \sim \mathcal{N}(0, \sigma_v^2 I)$$

Benefits: - Principled handling of uncertainty - Regularization from priors - Can incorporate side information

---

Bayesian Personalized Ranking (BPR)

For Implicit Feedback

Problem: With implicit data, we only have positive examples.

Approach: Learn to rank positives above negatives.

Assumption: User prefers items they interacted with over items they didn't.

The Loss

For triplet (user, positive item, negative item):

$$L = -\log \sigma(f(u, i^+) - f(u, i^-))$$

Where f is the prediction score (e.g., $u_u^T v_i$).

Training: Sample triplets and optimize.

Factorization Machines

Beyond Matrix Factorization

Matrix factorization only captures user-item interactions.

Factorization Machines generalize to any features:

$$f(x) = \mu + \sum_{j=1}^{d} w_j x_j + \sum_{j<k} (v_j \cdot v_k) x_j x_k$$

Where: - x: Feature vector (one-hot user, one-hot item, plus any other features) - v_j: Embedding for feature j

Advantages

- Can incorporate **side information**: User demographics, item attributes, context (time, location)
- Handles **cold start** better
- Same framework for different feature types

Connection to MF

When features are just user and item IDs:

$$f = \mu + b_u + c_i + u_u^T v_i$$

Exactly matrix factorization with biases!

The Cold Start Problem

The Challenge

New users or items have no interaction history.

Solutions

Content-based: Use features instead of collaborative signal - New user: Ask preferences, use demographics - New item: Use item attributes, description

Hybrid methods: Combine collaborative and content-based

Active learning: Ask strategic questions to new users

Transfer learning: Leverage data from related domains

Exploration-Exploitation Trade-off

The Problem

If we only recommend what users already like, they never discover new interests.

Counterfactual: Users might love items they never see!

Approaches

Multi-Armed Bandits: - **Thompson Sampling**: Sample from posterior, act greedily - **UCB**: Optimism in the face of uncertainty

Contextual Bandits: Personalized exploration based on user features

Diversity: Ensure recommendations aren't all the same type

Deep Learning for RecSys

Neural Collaborative Filtering

Replace dot product with neural network:

$$\hat{y}_{ui} = f_{neural}([u_u; v_i])$$

Benefit: Captures non-linear interactions.

Sequential Recommendations

Model user's sequence of interactions with RNN/Transformer:

$$h_t = f(x_t, h_{t-1})$$

Benefit: Captures temporal dynamics.

Two-Tower Models

Separate encoders for users and items: - Fast serving (pre-compute item embeddings) - Easy to add features

---

Evaluation Metrics

For Rating Prediction

- **RMSE**: $\sqrt{\frac{1}{N}\sum(y - \hat{y})^2}$
- **MAE**: $\frac{1}{N}\sum|y - \hat{y}|$

For Ranking

- **Precision@K**: Fraction of top-K recommendations that are relevant
- **Recall@K**: Fraction of relevant items in top-K
- **NDCG**: Accounts for position (higher rank = more important)
- **MAP**: Mean average precision

Offline vs. Online

Offline: Historical data, fast to compute but may not reflect real preferences.

Online (A/B testing): Real users, gold standard but expensive and slow.

Summary

Method	Key Idea	Best For
User-Based CF	Similar users □ similar items	Small, stable user base
Item-Based CF	Similar items	Most practical CF
Matrix Factorization	Low-rank approximation	General purpose
BPR	Learn to rank	Implicit feedback
Factorization Machines	Feature interactions	Side information
Neural Methods	Non-linear patterns	Large data, complex patterns

Practical Recommendations

1. **Start simple**: Matrix factorization with biases often hard to beat
2. **Add biases**: Critical for good performance
3. **Handle implicit correctly**: Use ranking losses, not rating losses
4. **Address cold start**: Hybrid methods or feature-based fallback
5. **Evaluate carefully**: Ranking metrics often more meaningful than RMSE

6. **Consider fairness**: Avoid filter bubbles, ensure diverse recommendations

ewpage

Part IV: Speech and Language Processing Notes

ewpage

Speech and Language Processing Notes

These notes are based on “Speech and Language Processing” by Dan Jurafsky and James H. Martin — the definitive textbook for Natural Language Processing. This guide makes NLP concepts accessible to undergraduates while building the foundation for advanced study.

What is Natural Language Processing?

NLP is the field of computer science focused on enabling computers to understand, interpret, and generate human language. It bridges linguistics, computer science, and machine learning.

Why is language hard for computers? - Language is **ambiguous** (bank of a river vs. bank for money) - Language is **context-dependent** (meaning changes with context) - Language is **creative** (infinite sentences from finite vocabulary) - Language has **implicit knowledge** (common sense, world knowledge)

Topics Covered

Chapter	Topic	What You’ll Learn
2	**Regular Expressions & Text Processing**	Pattern matching, tokenization, normalization
3	**N-Grams & Language Models**	Probabilistic models of word sequences
6	**Vector Semantics**	Word embeddings, similarity, Word2Vec
9	**Sequence Models**	RNNs, LSTMs, and attention mechanisms
10	**Encoder-Decoder Models**	Seq2Seq, machine translation
11	**Transfer Learning**	BERT, pre-training, fine-tuning

The Evolution of NLP

```
Rule-based    →    Statistical    →    Neural    →    Pre-trained
(1950s-1980s)      (1990s-2000s)     (2013-2018)    (2018-present)

Hand-crafted       Probabilistic      Deep learning   Transformers
grammars           models (HMMs,      (RNNs, LSTMs)   (BERT, GPT)
                   n-grams)
```

Core NLP Tasks

Understanding Language: - Text classification (spam detection, sentiment) - Named entity recognition (finding names, places, dates) - Parsing (sentence structure) - Semantic analysis (meaning extraction)

Generating Language: - Machine translation - Text summarization - Question answering - Dialogue systems

Prerequisites

- **Programming**: Python, basic data structures
- **Math**: Probability basics, linear algebra fundamentals
- **ML Basics**: Helpful but not strictly required

How to Use These Notes

1. **Start with fundamentals**: Regex and n-grams build the foundation
2. **Understand the progression**: From sparse to dense representations, from RNNs to Transformers
3. **Connect to practice**: Try implementing concepts in Python
4. **See the big picture**: Modern NLP combines all these ideas

Let's dive into the fascinating world of language and computation!

ewpage

Regular Expressions and Text Processing

Text processing is the foundation of NLP. Before we can analyze language, we need to handle raw text: finding patterns, breaking text into words, and normalizing variations.

The Big Picture

The Problem: Raw text is messy. We need systematic ways to: - Find patterns in text - Break text into meaningful units (tokens) - Handle variations (color vs. colour, ran vs. running)

The Tools: - **Regular expressions**: Powerful pattern matching - **Tokenization**: Splitting text into words/subwords - **Normalization**: Standardizing text formats

---

Regular Expressions

What Are Regular Expressions?

A **regex** is a language for specifying text patterns. Think of it as “find” on steroids.

Example: Find all email addresses in a document. - Without regex: Write complex code with many if statements - With regex: `/[\w.]+@[\w.]+\.\w+/`

Basic Patterns

Pattern	Meaning	Example Match
`/word/`	Exact match	“word”
`/[wW]ord/`	Either character	“word” or “Word”
`/[0-9]/`	Any digit	“0”, “5”, “9”
`/[a-z]/`	Any lowercase	“a”, “m”, “z”
`/[A-Z]/`	Any uppercase	“A”, “M”, “Z”

Negation with Caret

`[^...]` means “NOT these characters”:

Pattern	Meaning
`/[^A-Z]/`	Not an uppercase letter
`/[^0-9]/`	Not a digit
`/[^.]/`	Not a period

Note: Caret only means negation when it's the FIRST character inside brackets!

Quantifiers (How Many?)

Symbol	Meaning	Example
`?`	Zero or one	`/colou?r/` matches "color" and "colour"
`*`	Zero or more	`/a*/` matches "","a", "aaa"
`+`	One or more	`/[0-9]+/` matches "1", "42", "12345"
`{n}`	Exactly n	`/a{3}/` matches "aaa"
`{n,m}`	Between n and m	`/a{2,4}/` matches "aa", "aaa", "aaaa"
`{n,}`	At least n	`/a{2,}/` matches "aa", "aaa", …

Wildcards and Anchors

Wildcard: . matches any single character - `/beg.n/` matches "begin", "began", "begun"

Anchors (position markers):

Symbol	Meaning	Example
`^`	Start of line	`/^The/` matches lines starting with "The"
`$`	End of line	`/\.$/` matches lines ending with period
`\b`	Word boundary	`/\bthe\b/` matches "the" but not "there"

Grouping and Alternatives

Disjunction (`|`): Match either pattern - `/cat|dog/` matches "cat" or "dog"

Parentheses: Group patterns - `/gupp(y|ies)/` matches "guppy" or "guppies"

Character Classes (Shortcuts)

Shortcut	Meaning	Equivalent
`\d`	Digit	`[0-9]`
`\D`	Non-digit	`[^0-9]`
`\w`	Word character	`[a-zA-Z0-9_]`
`\W`	Non-word	`[^a-zA-Z0-9_]`
`\s`	Whitespace	`[ \t\n\r]`
`\S`	Non-whitespace	`[^ \t\n\r]`

Putting It Together

Find standalone “the” or “The”:

```
/(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)/
```

Breaking it down: - `(^|[^a-zA-Z])`: Start of line OR non-letter before - `[tT]he`: “the” or “The” - `([^a-zA-Z]|$)`: Non-letter after OR end of line

---

Words and Tokens

What Is a Word?

Seems simple, but it’s surprisingly tricky!

Challenges: - Contractions: Is “don’t” one word or two? - Hyphenation: Is “ice-cream” one word or two? - Languages without spaces: Chinese, Japanese - Multi-word expressions: “New York”, “kick the bucket”

Key Terminology

Term	Definition	Example
Token	An instance of a word/symbol	“the cat sat” has 3 tokens
Type	A unique word in vocabulary	“the cat sat on the mat” has 5 types
Lemma	Base/dictionary form	“runs”, “ran”, “running” □ “run”
Utterance	Spoken equivalent of sentence	What you actually say

Heap’s Law

Vocabulary size grows with corpus size, but sublinearly:

$$V = K \cdot N^{\beta}$$

Where: - V = vocabulary size (types) - N = corpus size (tokens) - β ≈ 0.5–0.7, K ≈ 10–100

Implication: You’ll always encounter new words!

Text Normalization

Three main steps: 1. **Tokenization**: Break into tokens 2. **Normalization**: Standardize format 3. **Segmentation**: Find sentence boundaries

Tokenization Approaches

Rule-Based (Penn Treebank): - Standard for English NLP - Specific rules for punctuation, contractions

Regex-Based (NLTK): - Flexible, customizable - Good for specific domains

Subword (BPE): - Data-driven - Handles unknown words gracefully

Byte Pair Encoding (BPE)

The Problem: What about words we've never seen? - Traditional tokenizers fail on "unigoogleable" - OOV (out-of-vocabulary) tokens hurt performance

The Solution: Learn tokens from data!

BPE Algorithm:

1. **Start**: Vocabulary = individual characters
2. **Count**: Find most frequent adjacent pair
3. **Merge**: Add merged pair to vocabulary
4. **Repeat**: Until target vocabulary size reached

Example:

```
Corpus: "low lower lowest"
Initial vocab: {l, o, w, e, r, s, t, _}

Step 1: Most frequent pair = "lo" → add "lo"
Step 2: Most frequent pair = "low" → add "low"
...
```

At test time: Apply merges in the same order they were learned.

Benefits: - No unknown words (can always fall back to characters) - Common words stay whole - Rare words split into meaningful pieces

---

Word Normalization

Case Folding

Convert to lowercase: - "Apple" □ "apple" - "HELLO" □ "hello"

Trade-off: Loses information! - "US" (country) vs. "us" (pronoun) - "Apple" (company) vs. "apple" (fruit)

Lemmatization

Reduce to base form: - "running", "ran", "runs" □ "run" - "better", "best" □ "good"

Requires: Understanding of morphology and often part-of-speech.

Stemming

Cruder approach — just chop suffixes: - “running” □ “run” - “happily” □ “happili” (imperfect!)

Porter Stemmer: Most common algorithm for English.

Edit Distance

The Problem

How similar are two strings?

Applications: - Spell checking - DNA sequence alignment - Plagiarism detection

Levenshtein Distance

Minimum number of **single-character edits**: - **Insert**: cat □ cat**s** - **Delete**: cats □ cat - **Substitute**: cat □ c**o**t

Dynamic Programming Solution

Build a matrix D where D[i,j] = distance between first i chars of string1 and first j chars of string2.

Initialization: - D[i,0] = i (delete all characters) - D[0,j] = j (insert all characters)

Recurrence:

```
D[i,j] = min(
    D[i-1,j] + 1,      # deletion
    D[i,j-1] + 1,      # insertion
    D[i-1,j-1] + cost  # substitution (cost=0 if match, 1 otherwise)
)
```

Example: Distance between “kitten” and “sitting” - Answer: 3 (k□s, e□i, +g)

Summary

Concept	Purpose	Key Tool
Regex	Find patterns	Pattern language
Tokenization	Split text	BPE, rules
Normalization	Standardize	Lemmatization, case folding
Edit Distance	Measure similarity	Dynamic programming

Practical Tips

1. **Always tokenize first** before any NLP task
2. **BPE** is the modern standard for neural models
3. **Be careful with normalization** — you might lose important information
4. **Regex takes practice** — use online testers to experiment!

ewpage

N-Grams and Language Models

Language models assign probabilities to sequences of words. They answer: “How likely is this sentence?” This fundamental capability underlies spell checking, machine translation, speech recognition, and text generation.

The Big Picture

Key Question: What’s the probability of a word sequence?

$$P(\text{”the cat sat on the mat”})$$

Why This Matters: - Spell checking: P(“the cat”) > P(“the kat”) - Machine translation: Choose more fluent translation - Speech recognition: Distinguish “recognize speech” from “wreck a nice beach” - Text generation: Sample likely continuations

---

Language Model Fundamentals

Joint Probability of Words

We want to compute:

$$P(w_1, w_2, ..., w_n)$$

Using the **chain rule** of probability:

$$P(w_1, w_2, ..., w_n) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_1, w_2) \times ... \times P(w_n|w_1, ..., w_{n-1})$$

More compactly:

$$P(w_1, w_2, ..., w_n) = \prod_{i=1}^{n} P(w_i|w_{1:i-1})$$

Problem: As sequences get longer, we need infinitely many parameters!

The Markov Assumption

Key insight: Approximate by using only recent history.

For a **bigram model** (2-gram):

$$P(w_n|w_1, w_2, ..., w_{n-1}) \approx P(w_n|w_{n-1})$$

Assumption: The next word depends only on the previous word.

N-Gram Models

Model	Conditioning Context	Example
Unigram	None	P(the)
Bigram	Previous 1 word	P(cat | the)
Trigram	Previous 2 words	P(sat | the cat)
4-gram	Previous 3 words	P(on | the cat sat)

Trade-off: Larger n = better context but more parameters and sparser data.

Estimating N-Gram Probabilities

Maximum Likelihood Estimation

Use **relative frequency** (counting):

Bigram probability:

$$P(w_n|w_{n-1}) = \frac{\text{count}(w_{n-1}, w_n)}{\text{count}(w_{n-1})}$$

Example: Computing P(sat | the) from a corpus: - Count “the sat” occurrences: 100 - Count “the ___” occurrences: 10,000 - P(sat | the) = 100/10,000 = 0.01

Handling Sentence Boundaries

Add special tokens: - **<s>**: Beginning of sentence (BOS) - **</s>**: End of sentence (EOS)

Example: “<s> the cat sat </s>”

This allows us to model: - P(the | <s>) — how likely is “the” to start a sentence? - P(</s> | sat) — how likely is “sat” to end a sentence?

Practical Note: Log Probabilities

Multiplying many small probabilities numerical underflow!

Solution: Work in log space:

$$\log(p_1 \times p_2) = \log(p_1) + \log(p_2)$$

Add log probabilities instead of multiplying probabilities.

Perplexity

What Is Perplexity?

The standard metric for evaluating language models.

Definition:

$$\text{PP}(W) = P(w_1, w_2, ..., w_n)^{-1/n}$$

Equivalently:

$$\text{PP}(W) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_{1:i-1})}}$$

Intuition: Weighted Branching Factor

Perplexity ≈ average number of equally likely choices at each step.

Example: - Perplexity of 100 ≈ model is choosing between 100 equally likely words - Perplexity of 10 ≈ model is choosing between 10 equally likely words

Lower perplexity = better model (more confident predictions).

Connection to Information Theory

Perplexity relates to **entropy**:

$$\text{PP}(W) = 2^{H(W)}$$

Where entropy H measures uncertainty:

$$H(W) = -\frac{1}{n} \log_2 P(w_1, ..., w_n)$$

Important Caveats

Perplexities are only comparable when: - Using the same vocabulary - Using the same test set

Adding rare words to vocabulary increases perplexity (more choices).

The Unknown Word Problem

The Problem

What if we see a word we've never seen before?

$$P(\text{unigoogleable}|w_{n-1}) = 0$$

Zero probability breaks everything: - Product becomes zero - Perplexity becomes infinite

Solution: <UNK> Token

1. Define a vocabulary (e.g., most frequent 50,000 words)
2. Replace all other words with <UNK>
3. Treat <UNK> as just another word

Training: "I saw a brachiosaurus" □ "I saw a <UNK>" **Testing**: Same replacement

Caveat: Smaller vocabulary = lower perplexity (fewer choices). Not always fair to compare!

Smoothing

The Zero Probability Problem

Even with <UNK>, we'll see **new n-grams**:

$$P(\text{cat}|\text{the green}) = 0 \text{ (if never seen together)}$$

Smoothing redistributes probability mass to unseen events.

Laplace (Add-One) Smoothing

Idea: Pretend we saw everything at least once.

Unigram:

$$P(w_i) = \frac{\text{count}(w_i) + 1}{N + V}$$

Bigram:

$$P(w_i|w_j) = \frac{\text{count}(w_j, w_i) + 1}{\text{count}(w_j) + V}$$

Where V is vocabulary size.

Problem: Add-1 is too aggressive — steals too much from seen events.

Solution: Add-k smoothing (k < 1, e.g., k = 0.01).

Backoff

Idea: If no evidence for trigram, use bigram. If no bigram, use unigram.

$$P_{BO}(w_i|w_{i-2}, w_{i-1}) = \begin{cases} P(w_i|w_{i-2}, w_{i-1}) & \text{if count > 0} \\ \alpha \cdot P_{BO}(w_i|w_{i-1}) & \text{otherwise} \end{cases}$$

Where α is a normalization factor.

Interpolation

Idea: Always mix all n-gram levels.

$$P(w_i|w_{i-2}, w_{i-1}) = \lambda_1 P(w_i) + \lambda_2 P(w_i|w_{i-1}) + \lambda_3 P(w_i|w_{i-2}, w_{i-1})$$

Where $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

Learn λ values from held-out data.

Kneser-Ney Smoothing

State-of-the-art for n-gram models.

Key insight: Use *continuation probability* — how likely is a word to appear in new contexts?

$$P_{KN}(w_i|w_{i-1}) = \frac{\max(\text{count}(w_{i-1}, w_i) - d, 0)}{\sum_v \text{count}(w_{i-1}, v)} + \lambda(w_{i-1}) P_{continuation}(w_i)$$

Intuition: - Words like “Francisco” have high count but appear after “San” — low continuation probability - Words like “the” appear in many contexts — high continuation probability

---

Efficiency Considerations

N-gram models can be huge! Practical tricks:

Technique	Purpose
Quantization	Store probabilities with fewer bits
Tries	Efficient storage and lookup
String hashing	Reduce memory for n-grams
Bloom filters	Fast membership testing
Stupid Backoff	Simple, fast approximation

Summary

Concept	Key Idea
Language Model	Assign probabilities to word sequences
N-gram	Approximate using last n-1 words
MLE	Estimate from counts (relative frequency)
Perplexity	Model evaluation metric (lower = better)
Smoothing	Handle unseen n-grams
Kneser-Ney	Best n-gram smoothing technique

The Bigger Picture

N-gram models are: - **Simple and interpretable** - **Capture local syntax** (word order) - **Fast** to train and use

But they have **limitations**: - **Fixed context** (can't capture long-range dependencies) - **Sparse data** (even trigrams need lots of data) - **No semantic understanding** (just counting patterns)

This motivates **neural language models** (covered in later chapters).

ewpage

Vector Semantics and Word Embeddings

How do we represent word meaning computationally? This chapter covers the evolution from sparse count-based vectors to dense neural embeddings — one of the most important advances in NLP.

The Big Picture

The Problem: Computers need numerical representations of words.

Key Insight (Distributional Hypothesis): > "You shall know a word by the company it keeps" — J.R. Firth

Words that appear in similar contexts have similar meanings.

The Evolution:

```
One-hot vectors → Count-based vectors → Neural embeddings
(sparse, no similarity)   (sparse, some similarity)   (dense, learned similarity)
```

Challenges of Lexical Semantics

Why is meaning hard?

Challenge	Example
Word forms	sing, sang, sung (same lemma "sing")
Polysemy	"bank" = river bank or financial bank
Synonymy	couch ≈ sofa (same meaning)
Relatedness	coffee ~ cup (not synonyms, but related)
Semantic frames	"A bought from B" ≈ "B sold to A"
Connotation	"slender" vs. "skinny" (same denotation, different feeling)

Vector Space Models

The Core Idea

Represent words as vectors in a high-dimensional space where: - **Similar words** are **close together** - **Dissimilar words** are **far apart**

Document Vectors (Term-Document Matrix)

	Doc1	Doc2	Doc3
cat	3	0	1
dog	2	4	0
pet	1	2	1

- **Rows**: Words (vocabulary of size V)
- **Columns**: Documents (D documents)
- **Cell**: Count of word in document

Use case: Information retrieval (find similar documents).

Word Vectors (Term-Term Matrix)

	cat	dog	pet	food
cat	-	15	20	8
dog	15	-	25	12
pet	20	25	-	10

- **Rows and Columns**: Words
- **Cell**: Co-occurrence count (how often words appear together)

Result: Each word is a V-dimensional vector.

Measuring Similarity

Cosine Similarity

Normalized dot product — measures angle between vectors:

$$\cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|} = \frac{\sum_i a_i b_i}{\sqrt{\sum_i a_i^2} \cdot \sqrt{\sum_i b_i^2}}$$

Interpretation: - cos = 1: Identical direction (most similar) - cos = 0: Perpendicular (unrelated) - cos = -1: Opposite direction (antonyms, in some cases)

Why cosine over Euclidean? - Handles different vector magnitudes - A long document and short document can still be similar

For Unit Vectors

When vectors are normalized (length 1):

$$||\vec{a} - \vec{b}||^2 = 2(1 - \cos\theta)$$

Euclidean distance and cosine become equivalent!

TF-IDF Weighting

Raw counts have problems: - Common words (“the”, “is”) dominate - Rare but meaningful words get drowned out

Term Frequency (TF)

How often does word appear in document?

Raw TF: $\text{tf}_{t,d} = \text{count}(t, d)$

Log TF (dampens large counts):

$$\text{tf}_{t,d} = \log(1 + \text{count}(t, d))$$

Inverse Document Frequency (IDF)

How rare is the word across documents?

$$\text{idf}_t = \log\left(\frac{N}{\text{df}_t}\right)$$

Where: - N = total number of documents - df_t = number of documents containing term t

Effect: Common words (low IDF) get downweighted.

TF-IDF

Combine both:

$$w_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$$

High TF-IDF: Word appears often in this document but rarely overall □ distinctive!

Pointwise Mutual Information (PMI)

The Intuition

Are two words appearing together more than we'd expect by chance?

$$\text{PMI}(x, y) = \log_2 \frac{P(x, y)}{P(x) \cdot P(y)}$$

Interpretation: - PMI > 0: Words co-occur more than expected (associated) - PMI = 0: Words co-occur as expected (independent) - PMI < 0: Words co-occur less than expected (avoid each other)

From Counts

$$\text{PMI}(x, y) = \log_2 \frac{\text{count}(x, y) \cdot N}{\text{count}(x) \cdot \text{count}(y)}$$

Positive PMI (PPMI)

Negative PMI values are unreliable (rare events).

$$\text{PPMI}(x, y) = \max(0, \text{PMI}(x, y))$$

From Sparse to Dense: Word2Vec

The Problem with Count Vectors

- **Very high dimensional** (vocabulary size)
- **Very sparse** (mostly zeros)
- **No generalization** between similar words

The Neural Solution

Learn **dense, low-dimensional** vectors (typically 100-300 dimensions).

Key properties: - Similar words have similar vectors - Relationships are captured geometrically

Static vs. Contextual Embeddings

Type	Same word = same vector?	Examples
Static	Yes	Word2Vec, GloVe, FastText
Contextual	No (depends on context)	ELMo, BERT, GPT

Skip-Gram with Negative Sampling (SGNS)

The most popular Word2Vec algorithm.

The Task

Given a target word, predict surrounding context words.

Example: “The quick **brown** fox jumps” - Target: “brown” - Context (window=2): “The”, “quick”, “fox”, “jumps”

Training Setup

1. **Positive examples**: (target, context) pairs from real text
2. **Negative examples**: (target, random_word) pairs — fake associations

The Objective

Maximize probability of real pairs, minimize probability of fake pairs:

$$L = \log \sigma(v_w \cdot v_c) + \sum_{i=1}^{k} \mathbb{E}_{c_i \sim P_n} [\log \sigma(-v_w \cdot v_{c_i})]$$

Where: - σ is sigmoid function - v_w is target word vector - v_c is context word vector - k is number of negative samples (typically 5-20)

Negative Sampling Distribution

Don’t sample uniformly — would get too many rare words.

$$P(w) \propto \text{freq}(w)^{0.75}$$

The 0.75 power smooths the distribution (gives rare words a better chance than pure frequency).

Two Embeddings Per Word

Each word has: - **Target embedding**: When it’s the center word - **Context embedding**: When it appears in context

Final embedding is often their sum or average.

Enhancements and Variations

FastText (Subword Embeddings)

Problem: What about unknown words like “ungooglable”?

Solution: Represent words as bag of character n-grams.

“where” □ {<wh, whe, her, ere, re>}

Word vector = sum of n-gram vectors.

Benefit: Can handle any word, even unseen ones!

GloVe (Global Vectors)

Combines advantages of count-based and neural methods.

Uses global co-occurrence statistics + optimization:

$$J = \sum_{i,j} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2$$

Often comparable to Word2Vec in practice.

Word Analogies

Famous Word2Vec property:

“king” - “man” + “woman” ≈ “queen”

Find word that completes analogy a:b :: a’:?

$$b' = \arg\min_x \text{distance}(x, b - a + a')$$

Works for: - Gender: king:queen :: man:woman - Capitals: Paris:France :: Tokyo:Japan - Tense: walking:walked :: swimming:swam

---

Bias in Word Embeddings

The Problem

Word embeddings learn biases present in training data.

Examples: - “doctor” closer to “man” than “woman” - “homemaker” closer to “woman” than “man” - Names associated with certain ethnic groups linked to negative words

Types of Harm

Allocation harm: System makes unfair decisions - Resume screening favoring male-associated names

Representation harm: Reinforces stereotypes - Search results, autocomplete suggestions

Mitigation Strategies

- Debias during training or post-hoc
- Careful data curation
- Evaluation for fairness

Summary

Representation	Pros	Cons
Count-based (TF-IDF)	Interpretable, simple	Sparse, high-dimensional
PMI	Captures associations	Sparse, noisy for rare words
Word2Vec	Dense, captures analogy	Static, no context
FastText	Handles OOV words	Still static
Contextual	Word sense disambiguation	Computationally expensive

Key Takeaways

1. **Words can be represented as vectors** in semantic space
2. **Distributional similarity** = semantic similarity
3. **Dense embeddings** outperform sparse for most tasks
4. **Context matters** — motivates contextual embeddings (BERT, etc.)
5. **Beware of biases** inherited from training data

ewpage

Sequence Architectures: RNNs, LSTMs, and Attention

Language is inherently sequential. This chapter covers neural architectures designed to process sequences: from basic RNNs to LSTMs to the attention mechanism that revolutionized NLP.

The Big Picture

The Problem: Language has dependencies across arbitrary distances. - "The **cat** that I saw yesterday **was** cute" (subject-verb agreement) - Standard feedforward networks have fixed input size

The Solution: Architectures with **memory** that process sequences step by step.

The Evolution:

```
FFNNs → RNNs → LSTMs/GRUs → Attention → Transformers
(fixed context)  (memory)  (better memory)  (direct connections)
```

Why Not Feedforward Networks?

Limitation: Fixed context window.

A bigram FFNN can only see the previous word. But language has long-range dependencies: - "The students who did well on the exam **were** happy" - Verb agrees with "students", not "exam"

We need: Variable-length context.

Recurrent Neural Networks (RNNs)

The Core Idea

Add a **recurrent connection** — the hidden state from the previous step feeds into the current step.

$$h_t = g(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

The hidden state h_t acts as memory!

Intuition

Think of reading a sentence word by word: - You update your understanding as each word arrives - Your current understanding depends on what you've read so far - That's exactly what h_t does

Training: Backpropagation Through Time (BPTT)

1. **Unroll** the network across time steps
2. **Forward pass**: Compute all hidden states and outputs
3. **Backward pass**: Compute gradients through the unrolled graph
4. **Update**: Sum gradients for shared weights

For long sequences: Use truncated BPTT (limit how far back gradients flow).

RNN Language Model

$$e_t = Ex_t \quad \text{(word embedding)}$$
$$h_t = g(W_{hh}h_{t-1} + W_{he}e_t)$$
$$P(w_{t+1}) = \text{softmax}(W_{hy}h_t)$$

Training: Teacher forcing — use true previous word, not predicted word.

Weight tying: Share parameters between input embedding E and output layer.

RNN Task Variants

Sequence Labeling (Many-to-Many, aligned)

Task: Label each token (NER, POS tagging).

```
Input:  John  loves  New   York
Output: B-PER O      B-LOC I-LOC
```

Predict at each step based on hidden state.

Sequence Classification (Many-to-One)

Task: Classify entire sequence (sentiment analysis).

```
Input:  This movie was great!
Output: POSITIVE
```

Use final hidden state (or pooled states) for classification.

Sequence Generation (One-to-Many or Many-to-Many)

Task: Generate text.

```
Input:  <BOS>
Output: The cat sat on the mat <EOS>
```

Autoregressive: Each output becomes next input.

RNN Architectures

Stacked RNNs

Multiple RNN layers:

```
Layer 3: h3_t = f(h3_{t-1}, h2_t)
Layer 2: h2_t = f(h2_{t-1}, h1_t)
Layer 1: h1_t = f(h1_{t-1}, x_t)
```

Benefit: Different abstraction levels at each layer.

Bidirectional RNNs

Process sequence both ways: - Forward: $\vec{h}_t$ (left to right) - Backward: $\overleftarrow{h}_t$ (right to left) - Combined: $h_t = [\vec{h}_t; \overleftarrow{h}_t]$

Benefit: Each position has access to full context.

Limitation: Can't use for autoregressive generation (need future that doesn't exist yet).

The Vanishing Gradient Problem

The Problem

Gradients shrink exponentially as they flow backward through time:

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial h_T} \cdot \prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}}$$

If $\frac{\partial h_t}{\partial h_{t-1}} < 1$ consistently □ **vanishing gradients**.

Consequence: RNNs struggle to learn long-range dependencies.

Why It Happens

- Sigmoid derivative: max 0.25
- Tanh derivative: max 1.0
- Repeated multiplication through many steps □ exponential decay

Solutions

1. **Gradient clipping** (for exploding gradients)
2. **Better architectures**: LSTM, GRU
3. **Skip connections**: Allow gradients to flow directly

LSTM (Long Short-Term Memory)

The Innovation

Add **explicit memory management** through **gates**.

Two types of state: - **Cell state** c_t: Long-term memory (conveyor belt) - **Hidden state** h_t: Working memory / output

The Three Gates

Gate	Purpose	Controls
Forget	What to erase from memory	f_t
Input	What new info to add	i_t
Output	What to expose as output	o_t

LSTM Equations

Forget gate (what to keep from old memory):

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

Input gate (how much of new info to add):

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

Candidate values (new info to potentially add):

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

Cell state update (the key equation!):

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

Output gate (what to output):

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

Hidden state:

$$h_t = o_t \odot \tanh(c_t)$$

Why LSTM Works

The cell state update is **additive**:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

Gradients can flow through unchanged (when forget gate ≈ 1).

GRU (Gated Recurrent Unit)

Simplified LSTM with fewer parameters: - Combines forget and input gates into **update gate** - No separate cell state

Often performs comparably to LSTM with less computation.

Attention Mechanism

The Bottleneck Problem

In encoder-decoder models, all information must pass through a fixed-size vector.

Problem: Information gets lost for long sequences.

The Attention Solution

Let the decoder **look at all encoder states** when making each prediction.

How Attention Works

For each decoder step:

1. **Score**: Compute similarity between decoder state and each encoder state

$$e_{ij} = \text{score}(s_{i-1}, h_j)$$

2. **Normalize**: Convert scores to weights (softmax)

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$$

3. **Combine**: Weighted sum of encoder states

$$c_i = \sum_j \alpha_{ij} h_j$$

4. **Use**: Context vector informs prediction

$$s_i = f(s_{i-1}, y_{i-1}, c_i)$$

Scoring Functions

Type	Formula
Dot product	$s^T h$
Scaled dot product	$\frac{s^T h}{\sqrt{d}}$
MLP	$v^T \tanh(W_1 s + W_2 h)$

Benefits of Attention

1. **Long-range dependencies**: Direct connections regardless of distance
2. **Interpretability**: Attention weights show what the model focuses on
3. **Alignment**: Helpful for translation (which source words map to which target words)

Self-Attention and Transformers

From Attention to Self-Attention

Regular attention: Query from decoder, keys/values from encoder.

Self-attention: Query, keys, values all from same sequence.

$$\text{output}_i = \text{Attention}(x_i, (x_1, x_1), (x_2, x_2), ..., (x_n, x_n))$$

Each position attends to all positions in the same sequence.

Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where: - Q = queries (what I'm looking for) - K = keys (what I offer for matching) - V = values (what I actually provide) - $\sqrt{d_k}$ scaling prevents softmax saturation

Multi-Head Attention

Run multiple attention operations in parallel:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

$$\text{MultiHead} = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

Benefit: Each head can capture different types of relationships.

Positional Encoding

Attention is **permutation invariant** — doesn't know word order!

Solution: Add position information to embeddings.

Sinusoidal encoding:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d})$$

BERT Architecture

Base model: - 12 attention heads - 12 layers - 768 hidden size (12 × 64) - 110M parameters

---

Summary

Architecture	Memory	Long-range	Parallelizable
RNN	Hidden state	Limited	No
LSTM	Cell + hidden	Better	No
Attention RNN	+ context	Good	Partially
Transformer	Attention only	Excellent	Yes

Key Takeaways

1. **RNNs** process sequences with memory but struggle with long dependencies
2. **LSTMs** use gates to control information flow, solving vanishing gradients
3. **Attention** provides direct connections between any positions
4. **Transformers** replace recurrence with pure attention, enabling parallelism
5. Modern NLP is dominated by **Transformer-based models** (BERT, GPT, etc.)

ewpage

Encoder-Decoder Models

Encoder-decoder (seq2seq) models transform one sequence into another. They power machine translation, summarization, question answering, and many other NLP tasks.

The Big Picture

The Problem: Input and output sequences have different lengths and structures.

Example (Translation): - English: “The cat sat on the mat” (6 words) - German: “Die Katze saß auf der Matte” (6 words, different order) - Japanese: □□□□□□□□□□□ (different structure entirely)

The Solution: 1. **Encoder**: Compress input into a representation 2. **Decoder**: Generate output from that representation

---

Encoder-Decoder Architecture

The Two Components

```
Input Sequence → [ENCODER] → Context Vector → [DECODER] → Output Sequence
```

Encoder: - Processes input sequence - Produces contextualized hidden states - Creates a “summary” of the input

Decoder: - Uses encoder output as initial context - Generates output tokens one at a time - Autoregressive: each output depends on previous outputs

---

Sequence-to-Sequence with RNNs

Encoder

Process input token by token:

$$h_t^{enc} = f(h_{t-1}^{enc}, x_t)$$

The final hidden state h_T^{enc} summarizes the entire input.

Context Vector

Simple approach: Use final encoder hidden state.

$$c = h_T^{enc}$$

Problem: All information must squeeze through this bottleneck!

Decoder

Initialize with context, generate autoregressively:

$$h_t^{dec} = f(h_{t-1}^{dec}, y_{t-1}, c)$$

$$P(y_t) = \text{softmax}(W h_t^{dec})$$

Training: Teacher Forcing

During training, use **ground truth** previous tokens, not predicted ones.

Without teacher forcing: Errors compound (predicted mistake □ more mistakes). **With teacher forcing**: More stable training, faster convergence.

The drawback: Exposure bias — model never sees its own mistakes during training.

The Attention Solution

The Bottleneck Problem

As sequences get longer, the fixed-size context vector struggles to capture everything.

Dynamic Context

Instead of one context vector, compute a **different context for each decoder step**:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j^{enc}$$

Where α_{ij} are attention weights — how much should decoder step i focus on encoder position j?

Computing Attention Weights

1. **Score** each encoder state against decoder state:

$$e_{ij} = \text{score}(s_{i-1}^{dec}, h_j^{enc})$$

2. **Normalize** to get weights:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

3. **Combine** to get context:

$$c_i = \sum_j \alpha_{ij} h_j^{enc}$$

4. **Decode** using context:

$$s_i^{dec} = f(s_{i-1}^{dec}, y_{i-1}, c_i)$$

Benefits of Attention

Benefit	Explanation
Long sequences	No information bottleneck
Alignment	Learns which source words map to which target words
Interpretability	Can visualize what the model focuses on
Gradient flow	Direct paths for gradients

Transformer Encoder-Decoder

Key Difference: Cross-Attention

The decoder has three types of attention: 1. **Self-attention** on encoder (bidirectional) 2. **Masked self-attention** on decoder (causal — can't see future) 3. **Cross-attention**: Queries from decoder, Keys/Values from encoder

Cross-Attention Mechanism

$$\text{CrossAttn}(Q^{dec}, K^{enc}, V^{enc}) = \text{softmax}\left(\frac{Q^{dec}(K^{enc})^T}{\sqrt{d}}\right) V^{enc}$$

Decoder queries look up relevant information from encoder.

Tokenization for Seq2Seq

The Challenge

Different languages have different: - Writing systems - Word boundaries - Vocabulary sizes

Subword Tokenization

Use **BPE** or **WordPiece** for both languages: - Handles rare words gracefully - Shares subwords across similar languages - Reduces vocabulary size

---

Evaluation Metrics

Human Evaluation (Gold Standard)

Adequacy: Is the meaning preserved? - 1 = None, 5 = All meaning captured

Fluency: Is it grammatically correct and natural? - 1 = Incomprehensible, 5 = Native quality

Problem: Expensive, slow, not reproducible.

Automatic Metrics

BLEU (Bilingual Evaluation Understudy)

The classic MT metric.

Core idea: Count n-gram matches between output and reference.

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

Where: - p_n = precision for n-grams (typically n=1,2,3,4) - w_n = weights (usually uniform: 0.25 each) - BP = brevity penalty (penalizes short translations)

Brevity Penalty:

$$BP = \min\left(1, \exp\left(1 - \frac{r}{c}\right)\right)$$

Where r = reference length, c = candidate length.

BLEU ranges: 0 to 100 (100 = perfect match).

Limitations: - Doesn't capture meaning (just surface n-grams) - One reference may miss valid alternatives - Insensitive to word order issues

chrF (Character F-Score)

Uses character n-grams instead of word n-grams.

Benefits: - Works for languages without clear word boundaries - More robust to morphological variation - Often correlates better with human judgment

BERTScore

Uses neural embeddings for semantic matching.

1. Embed each token in reference and hypothesis using BERT
2. Greedily match tokens by cosine similarity
3. Compute precision, recall, F1 from matches

Benefits: - Captures semantic similarity, not just surface form - “automobile” matches “car” even though different words

---

Decoding Strategies

The Challenge

At each step, we have a probability distribution over the entire vocabulary.

Goal: Find the most likely complete sequence.

Problem: Exhaustive search is intractable (V^T possibilities).

Greedy Decoding

Pick highest probability token at each step:

$$y_t = \arg\max_y P(y|y_{<t}, x)$$

Pros: Fast, simple. **Cons**: Locally optimal ≠ globally optimal. High-probability first word might lead to low-probability continuation.

Beam Search

Keep top-K hypotheses at each step:

1. Start with K copies of <BOS>
2. Expand each hypothesis with all possible next tokens
3. Keep top K by total probability
4. Repeat until all hypotheses end with <EOS>
5. Return highest-scoring complete hypothesis

Beam width K: - K=1 is greedy decoding - K=5-10 typical for translation - Larger K = better but slower

Length normalization (prevent bias toward short sequences):

$$\text{score} = \frac{\log P(Y|X)}{|Y|^\alpha}$$

Where α ≈ 0.6-0.7.

Sampling Strategies (for Generation)

For creative text generation, we want **diversity**, not just the most likely output.

Top-K Sampling: 1. Keep only top K most probable tokens 2. Redistribute probability among them 3. Sample from this truncated distribution

Top-P (Nucleus) Sampling: 1. Sort tokens by probability 2. Keep smallest set with cumulative probability > p 3. Sample from this set

Temperature (before softmax):

$$P(y) = \frac{\exp(z_y/T)}{\sum_j \exp(z_j/T)}$$

- T < 1: More peaked (confident, less diverse)
- T = 1: Original distribution
- T > 1: Flatter (more random, more diverse)

Summary

Component	Purpose
Encoder	Compress input to representation
Decoder	Generate output autoregressively
Attention	Dynamic context at each step
Cross-attention	Transformer way of connecting encoder to decoder
Beam search	Better than greedy, tractable search
BLEU/BERTScore	Automatic evaluation

Key Takeaways

1. **Encoder-decoder** is the standard architecture for sequence transduction
2. **Attention** solves the bottleneck problem and provides interpretability
3. **Evaluation is hard** — automatic metrics are imperfect proxies for quality
4. **Decoding strategy matters** — beam search for accuracy, sampling for diversity

ewpage

Transfer Learning and Pre-trained Models

Transfer learning has revolutionized NLP. Pre-train a large model on massive text data, then fine-tune for specific tasks. This chapter covers BERT and the pre-train/fine-tune paradigm.

The Big Picture

The Old Way: Train a model from scratch for each task. - Requires lots of labeled data - Each task starts from zero

The New Way: Pre-train □ Fine-tune. - Pre-train once on huge unlabeled corpus - Fine-tune with small labeled dataset per task - Transfer linguistic knowledge across tasks

---

Key Concepts

Contextual Embeddings

Static embeddings (Word2Vec): Same vector for “bank” regardless of context.

Contextual embeddings (BERT): Different vector for “river bank” vs. “bank account”.

The same word gets different representations based on surrounding words.

The Pre-train □ Fine-tune Paradigm

```
[MASSIVE UNLABELED TEXT] → Pre-training → [GENERAL LANGUAGE MODEL]
                                                        ↓
[SMALL LABELED DATA] → Fine-tuning → [TASK-SPECIFIC MODEL]
```

Pre-training: Self-supervised learning on vast text (books, Wikipedia, web).

Fine-tuning: Supervised learning on task-specific data.

Language Model Types

Type	Direction	Example	Best For
Causal	Left-to-right	GPT	Generation
Bidirectional	Both directions	BERT	Understanding
Encoder-Decoder	Both + generation	T5	Both

Bidirectional Transformers (BERT)

Why Bidirectional?

For many tasks, we can see the entire input at once.

Causal models (GPT): Can only see left context.

```
"The cat sat on the [???]" - what comes next?
```

Bidirectional models (BERT): See full context.

```
"The cat sat on the [MASK]" - what's missing?
```

BERT Architecture

Self-attention across entire sequence:

$$\text{Attention} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

BERT Base: - Vocabulary: 30K subwords (WordPiece) - Hidden size: 768 (12 heads × 64 dims) - Layers: 12 - Parameters: 110M - Max sequence: 512 tokens

Compute note: Attention is O(n²) in sequence length — limits max length.

Pre-training Objectives

Masked Language Modeling (MLM)

The task: Predict randomly masked tokens.

```
Input:  The cat [MASK] on the mat
Target: sat
```

Masking strategy (for 15% of tokens): - 80%: Replace with [MASK] - 10%: Replace with random word - 10%: Keep original

Why this mix? - [MASK] never appears at fine-tuning □ train on real words too - Random replacement adds noise, prevents overfitting - Keeping some originals helps learn bidirectional context

Span Masking (SpanBERT)

Mask contiguous spans instead of random tokens.

```
Input:  The [MASK] [MASK] [MASK] the mat
Target: cat sat on
```

Benefits: - Better for tasks requiring span understanding (QA, NER) - Span Boundary Objective: Predict span from boundary tokens

Next Sentence Prediction (NSP)

The task: Are these sentences adjacent in the original text?

```
[CLS] The cat sat [SEP] It was happy [SEP] → IsNext
[CLS] The cat sat [SEP] I like pizza [SEP] → NotNext
```

Training data: 50% real pairs, 50% random pairs.

Note: Later models (RoBERTa, ALBERT) found NSP less helpful than expected.

Pre-training Data

BERT was trained on: - **BooksCorpus**: 800M words - **English Wikipedia**: 2.5B words

Later models use more: - CommonCrawl (filtered web text) - News articles - Code repositories

Compute requirement: Days to weeks on TPU/GPU clusters.

Fine-tuning

The Basic Recipe

1. Load pre-trained model
2. Add task-specific **classification head** (usually 1-2 layers)
3. Train on labeled data with small learning rate

Fine-tuning Strategies

Strategy	What's Updated	Best For
Full fine-tuning	All parameters	Lots of data, maximum performance
Feature extraction	Only head	Very small data
Adapter tuning	Small inserted modules	Efficient multi-task
Prompt tuning	Soft prompts only	Very large models

Hyperparameters

Learning rate: 2e-5 to 5e-5 (much smaller than training from scratch!)

Epochs: 2-4 (often sufficient)

Batch size: 16-32

Task-Specific Architectures

Sequence Classification

Task: Classify entire input (sentiment, topic).

```
Input:  [CLS] I loved this movie [SEP]
Output: Use [CLS] representation → linear → softmax → class
```

Sentence Pair Classification

Task: Classify relationship between two texts (NLI, similarity).

```
Input:  [CLS] Premise text [SEP] Hypothesis text [SEP]
Output: [CLS] representation → linear → entailment/contradiction/neutral
```

Token Classification (NER, POS)

Task: Label each token.

```
Input:  [CLS] John lives in New York [SEP]
Labels:       B-PER O     O  B-LOC I-LOC
```

Each token gets its own classification head output.

WordPiece handling: - Training: Expand labels to all subword tokens - Evaluation: Use label of first subword

Span Prediction (QA)

Task: Find answer span in context.

```
Input:  [CLS] Where is Paris? [SEP] Paris is in France [SEP]
Output: Predict start position (index 5: "Paris")
        Predict end position (index 8: "France")
```

Two classifiers: one for start, one for end position.

Modern Variants

RoBERTa (Robustly Optimized BERT)

Key changes: - Remove NSP objective - Larger batches, more data - Dynamic masking (different masks each epoch)

ALBERT (A Lite BERT)

Parameter reduction: - Factorized embedding parameters - Cross-layer parameter sharing - Sentence order prediction instead of NSP

DistilBERT

Knowledge distillation: - 40% smaller, 60% faster - 97% of BERT performance

---

Summary

Concept	Key Point
Contextual embeddings	Same word □ different vectors in different contexts
Pre-training	Self-supervised on massive text
Fine-tuning	Task-specific with small labeled data
MLM	Predict masked tokens (BERT's main objective)
[CLS] token	Aggregate representation for classification
[SEP] token	Separate segments in input

The Revolution

Transfer learning changed NLP:

Before	After
Train from scratch per task	Pre-train once, fine-tune many times
Need lots of labeled data	Works with small labeled data
Shallow features	Deep contextual understanding
Task-specific architectures	One architecture, many tasks

Practical Tips

1. **Start with pre-trained models** — rarely worth training from scratch
2. **Try multiple learning rates** — this is the most important hyperparameter
3. **Don't over-fine-tune** — 2-4 epochs is often enough
4. **Consider model size** — DistilBERT for production, BERT-large for best accuracy
5. **Domain matters** — SciBERT for science, BioBERT for biomedical, etc.