

Machine Learning, Artificial Intelligence & Data Science Notes (DRAFT)

Version 0.0.1

Francis Tseng

Contents

| | |
|--|-----------|
| Introduction | 21 |
| I Foundations | 23 |
| 1 Functions | 25 |
| 1.0.1 Identity functions | 25 |
| 1.0.2 The inverse of a function | 26 |
| 1.0.3 Surjective functions | 26 |
| 1.0.4 Injective functions | 26 |
| 1.0.5 Surjective & injective functions | 26 |
| 1.0.6 Convex and non-convex functions | 26 |
| 1.1 References | 27 |
| 2 Linear Algebra | 29 |
| 2.1 Vectors | 29 |
| 2.1.1 Real coordinate spaces | 29 |
| 2.1.2 Column and row vectors | 30 |
| 2.1.3 Transposing a vector | 30 |
| 2.1.4 Vectors operations | 30 |
| 2.1.5 Norms | 31 |
| 2.1.6 Unit vectors | 33 |
| 2.1.7 Angles between vectors | 34 |
| 2.1.8 Perpendicular vectors | 34 |
| 2.1.9 Normal vectors | 35 |
| 2.1.10 Orthonormal vectors | 35 |
| 2.1.11 Additional vector operations | 35 |

| | | |
|--------|---|----|
| 2.2 | Linear Combinations | 36 |
| 2.2.1 | Parametric representations of lines | 36 |
| 2.2.2 | Linear combinations | 38 |
| 2.2.3 | Spans | 39 |
| 2.2.4 | Linear independence | 39 |
| 2.3 | Matrices | 42 |
| 2.3.1 | Matrix operations | 42 |
| 2.3.2 | The identity matrix | 45 |
| 2.3.3 | Diagonal matrices | 45 |
| 2.3.4 | Some properties of matrices | 46 |
| 2.3.5 | Matrix inverses | 46 |
| 2.3.6 | Matrix determinants | 46 |
| 2.3.7 | Transpose of a matrix | 48 |
| 2.3.8 | Symmetric matrices | 48 |
| 2.3.9 | The Trace | 50 |
| 2.3.10 | Orthogonal matrix | 50 |
| 2.3.11 | Adjoints | 51 |
| 2.4 | Subspaces | 51 |
| 2.4.1 | Spans and subspaces | 52 |
| 2.4.2 | Basis of a subspace | 53 |
| 2.4.3 | Dimension of a subspace | 53 |
| 2.4.4 | Nullspace of a matrix | 53 |
| 2.4.5 | Columnspace | 55 |
| 2.4.6 | Rank | 56 |
| 2.4.7 | The standard basis | 56 |
| 2.4.8 | Orthogonal compliments | 56 |
| 2.4.9 | Coordinates with respect to a basis | 57 |
| 2.4.10 | Orthonormal bases | 60 |
| 2.5 | Transformations | 61 |
| 2.5.1 | Linear transformations | 61 |
| 2.5.2 | Kernels | 62 |
| 2.6 | Images | 63 |
| 2.6.1 | Image of a subset of a domain | 63 |

| | | |
|----------|--|-----------|
| 2.6.2 | Image of a subspace | 64 |
| 2.6.3 | Image of a transformation | 64 |
| 2.6.4 | Preimage of a set | 64 |
| 2.7 | Projections | 65 |
| 2.7.1 | Projections onto subspaces | 67 |
| 2.8 | Identifying transformation properties | 68 |
| 2.8.1 | Determining if a transformation is surjective | 68 |
| 2.8.2 | Determining if a transformation is injective | 68 |
| 2.8.3 | Determining if a transformation is invertible | 68 |
| 2.8.4 | Inverse transformations of linear transformations | 69 |
| 2.9 | Eigenvalues and Eigenvectors | 69 |
| 2.9.1 | Properties of eigenvalues and eigenvectors | 70 |
| 2.9.2 | Diagonalizable matrices | 71 |
| 2.9.3 | Eigenvalues & eigenvectors of symmetric matrices | 71 |
| 2.9.4 | Eigenspace | 72 |
| 2.9.5 | Eigenbasis | 72 |
| 3 | Calculus | 73 |
| 3.1 | Differentiation | 73 |
| 3.1.1 | Formal definition of a derivative | 74 |
| 3.1.2 | Notation | 74 |
| 3.1.3 | Differentiation rules | 74 |
| 3.1.4 | Higher order derivatives | 76 |
| 3.1.5 | Explicit differentiation | 76 |
| 3.1.6 | Implicit differentiation | 77 |
| 3.1.7 | Derivatives of trigonometric functions | 78 |
| 3.1.8 | Derivatives of exponential and logarithmic functions | 78 |
| 3.1.9 | Extreme Value Theorem | 78 |
| 3.1.10 | Rolle's Theorem | 80 |
| 3.1.11 | Mean Value Theorem | 80 |
| 3.1.12 | L'Hopital's Rule | 80 |
| 3.2 | Integration | 81 |
| 3.2.1 | Definite integral | 81 |
| 3.2.2 | Basic properties of the integral | 82 |

| | | |
|--------|--|----|
| 3.2.3 | Mean Value Theorem for Integration | 82 |
| 3.2.4 | The fundamental theorem of calculus | 83 |
| 3.2.5 | Indefinite integral | 83 |
| 3.2.6 | Improper integrals | 84 |
| 3.3 | Multivariable Calculus | 86 |
| 3.3.1 | Integration | 86 |
| 3.3.2 | Partial derivatives | 88 |
| 3.3.3 | Directional derivatives | 90 |
| 3.3.4 | Gradients | 90 |
| 3.3.5 | Scalar and vector fields | 91 |
| 3.3.6 | Divergence | 92 |
| 3.3.7 | Curl | 93 |
| 3.3.8 | The Hessian | 93 |
| 3.3.9 | The Jacobian | 93 |
| 3.3.10 | Optimization with eigenvalues | 94 |
| 3.4 | Differential Equations | 94 |
| 3.4.1 | Solving simple differential equations | 95 |
| 3.4.2 | Basic first order differential equations | 95 |
| 3.5 | References | 98 |

| | | |
|----------|---|-----------|
| 4 | Probability | 99 |
| 4.1 | Probability space | 99 |
| 4.2 | Random Variables | 100 |
| 4.3 | Joint and disjoint probabilities | 101 |
| 4.4 | Conditional Probability | 101 |
| 4.5 | Independence | 102 |
| 4.5.1 | Conditional Independence | 103 |
| 4.6 | The Chain Rule of Probability | 103 |
| 4.7 | Combinations and Permutations | 104 |
| 4.7.1 | Permutations | 104 |
| 4.7.2 | Combinations | 104 |
| 4.7.3 | Combinations, permutations, and probability | 105 |
| 4.8 | Probability Distributions | 105 |
| 4.8.1 | Probability Mass Functions (PMF) | 106 |

| | | |
|--------|---|-----|
| 4.8.2 | Probability Density Functions (PDF) | 106 |
| 4.8.3 | Distribution Patterns | 107 |
| 4.9 | Cumulative Distribution Functions (CDF) | 109 |
| 4.9.1 | Discrete random variables | 109 |
| 4.9.2 | Continuous random variables | 110 |
| 4.9.3 | Using CDFs | 110 |
| 4.9.4 | Survival function | 112 |
| 4.10 | Expected Values | 113 |
| 4.10.1 | Discrete random variables | 113 |
| 4.10.2 | Continuous random variables | 114 |
| 4.10.3 | The expectation rule | 114 |
| 4.10.4 | Properties of expectations | 114 |
| 4.11 | Variance | 115 |
| 4.11.1 | Covariance | 115 |
| 4.12 | Common Probability Distributions | 116 |
| 4.12.1 | Probability mass functions | 116 |
| 4.12.2 | Probability density functions | 120 |
| 4.13 | Multiple random variables | 126 |
| 4.13.1 | Conditional distributions | 127 |
| 4.13.2 | Multivariate Gaussian | 128 |
| 4.14 | Bayes' Theorem | 131 |
| 4.14.1 | Intuition | 131 |
| 4.14.2 | A Visual Explanation | 131 |
| 4.14.3 | An Example Bayes' Problem | 132 |
| 4.14.4 | Solving the problem with Bayes' Theorem | 133 |
| 4.14.5 | Another Example | 133 |
| 4.14.6 | Naive Bayes | 134 |
| 4.15 | Entropy | 135 |
| 4.16 | The log trick | 136 |
| 4.17 | References | 136 |

| | |
|--|------------|
| 5 Statistics | 137 |
| 5.0.1 Notation | 137 |
| 5.1 Descriptive Statistics | 137 |
| 5.1.1 Scales of Measurement | 137 |
| 5.1.2 Averages | 138 |
| 5.1.3 Population vs Sample | 139 |
| 5.1.4 Independent and Identically Distributed | 140 |
| 5.1.5 The Law of Large Numbers (LLN) | 141 |
| 5.1.6 Regression to the mean | 141 |
| 5.1.7 Central Limit Theorem (CLT) | 141 |
| 5.1.8 Dispersion (Variance and Standard Deviation) | 141 |
| 5.1.9 Moments | 143 |
| 5.1.10 Covariance | 144 |
| 5.1.11 Correlation | 145 |
| 5.1.12 Degrees of Freedom | 147 |
| 5.1.13 Time Series Analysis | 148 |
| 5.1.14 Survival Analysis | 148 |
| 5.2 Inferential Statistics | 149 |
| 5.2.1 Error | 150 |
| 5.2.2 Estimates and estimators | 150 |
| 5.2.3 Point Estimation | 151 |
| 5.2.4 Nuisance Parameters | 152 |
| 5.2.5 Confidence Intervals | 152 |
| 5.2.6 Kernel Density Estimates | 153 |
| 5.3 Experimental Statistics | 154 |
| 5.3.1 Statistical Power | 155 |
| 5.3.2 Sample Selection | 155 |
| 5.3.3 The Null Hypothesis | 156 |
| 5.3.4 Type 1 Errors | 156 |
| 5.3.5 P Values | 156 |
| 5.3.6 The Base Rate Fallacy | 157 |
| 5.3.7 False Discovery Rate | 159 |
| 5.3.8 Alpha Level | 159 |

| | | |
|----------|---|------------|
| 5.3.9 | The Benjamini-Hochberg Procedure | 160 |
| 5.3.10 | Sum of Squares | 161 |
| 5.3.11 | Statistical Tests | 161 |
| 5.3.12 | Effect Size | 164 |
| 5.3.13 | Reliability | 165 |
| 5.3.14 | Agreement | 166 |
| 5.4 | Handling Data | 166 |
| 5.4.1 | Transforming data | 166 |
| 5.4.2 | Dealing with missing data | 167 |
| 5.4.3 | Resampling | 167 |
| 5.5 | References | 168 |
| 6 | Bayesian Statistics | 171 |
| 6.0.1 | Frequentist vs Bayesian approaches | 172 |
| 6.1 | Bayes' Rule | 172 |
| 6.2 | Choosing a prior distribution | 173 |
| 6.2.1 | Conjugate priors | 174 |
| 6.2.2 | Sensitivity Analysis | 176 |
| 6.2.3 | Empirical Bayes | 176 |
| 6.3 | Markov Chain Monte Carlo (MCMC) | 176 |
| 6.3.1 | Monte Carlo Integration | 176 |
| 6.3.2 | Markov Chains | 177 |
| 6.3.3 | Markov Chain Monte Carlo | 178 |
| 6.4 | Bayesian point estimates | 182 |
| 6.5 | Credible Intervals (Credible Regions) | 183 |
| 6.6 | Bayesian Regression | 183 |
| 6.7 | A Bayesian example | 184 |
| 6.7.1 | References | 186 |
| 7 | Probabilistic Graphical Models | 189 |
| 7.1 | Bayesian Networks | 189 |
| 7.2 | Hidden Markov Models (HMM) | 191 |
| 7.2.1 | Example | 192 |
| 7.3 | References | 192 |

| | |
|----------|----|
| CONTENTS | 10 |
|----------|----|

| | |
|--|------------|
| 8 Optimization | 193 |
| 8.1 Gradient Descent | 194 |
| 8.1.1 Stochastic gradient descent (SGD) | 195 |
| 8.1.2 Learning rates | 196 |
| 8.2 Simulated Annealing | 196 |
| 8.3 Nelder-Mead (aka Simplex or Amoeba optimization) | 197 |
| 8.4 Particle Swarm Optimization | 198 |
| 8.5 Genetic Algorithms | 199 |
| 8.6 Derivative-Free Optimization | 200 |
| 8.7 References | 200 |

| | |
|----------------------------|------------|
| II Machine Learning | 201 |
|----------------------------|------------|

| | |
|---|------------|
| 9 Machine Learning | 203 |
| 9.1 Representation vs Learning: | 204 |
| 9.2 References | 204 |
| 9.3 Supervised Learning | 204 |
| 9.4 Unsupervised Learning | 205 |
| 9.5 Other types of learning | 205 |
| 9.5.1 Semi-supervised Learning | 205 |
| 9.5.2 Active Learning | 205 |
| 9.5.3 Reinforcement Learning | 205 |
| 9.6 Representation | 205 |
| 9.6.1 Deep Learning | 206 |
| 9.6.2 References | 206 |
| 9.7 Terminology | 206 |
| 9.8 Optimization | 206 |
| 9.9 Linear Regression with One Variable | 207 |
| 9.9.1 How are the parameters determined? | 208 |
| 9.10 Gradient Descent | 210 |
| 9.10.1 Gradient Descent for Univariate Linear Regression | 210 |
| 9.11 Linear Regression with Multiple Variables | 211 |
| 9.12 Gradient descent with Multivariate Linear Regression | 212 |

| | |
|---|-----|
| 9.12.1 Feature Scaling | 212 |
| 9.12.2 Choosing the Learning Rate α | 213 |
| 9.13 Example implementation of linear regression with gradient descent | 214 |
| 9.14 Polynomial Regression | 215 |
| 9.15 Feature Engineering | 216 |
| 9.16 Normal Equation | 216 |
| 9.16.1 Deciding between Gradient Descent and the Normal Equation | 217 |
| 9.17 Classification | 218 |
| 9.17.1 Logistic Regression | 218 |
| 9.17.2 Confusion Matrices | 221 |
| 9.18 Overfitting | 222 |
| 9.19 Regularization | 223 |
| 9.19.1 Regularized Linear Regression | 224 |
| 9.19.2 Regularized Logistic Regression | 224 |
| 9.20 Discriminative vs Generative learning algorithms | 225 |
| 9.21 Neural Networks | 226 |
| 9.21.1 Neural networks and overfitting | 226 |
| 9.21.2 Determining the number of hidden layers to use | 226 |
| 9.22 Metrics | 226 |
| 9.22.1 Area Under Curve (AUC) | 226 |
| 9.23 Strategies for Applying Machine Learning | 227 |
| 9.23.1 What if your algorithm doesn't perform well? What should you try next? | 227 |
| 9.23.2 Machine learning diagnostics | 227 |
| 9.23.3 Evaluating a hypothesis | 227 |
| 9.23.4 Choosing a good λ for regularization | 228 |
| 9.23.5 Learning curves | 229 |
| 9.24 Machine Learning System Design | 229 |
| 9.25 Unsupervised Learning | 231 |
| 9.25.1 K-Means Clustering Algorithm | 231 |
| 9.25.2 Hierarchical Agglomerative Clustering | 232 |
| 9.26 Support Vector Machines | 233 |
| 9.26.1 Kernels | 235 |
| 9.27 Dimensionality Reduction | 243 |

| | |
|--|-----|
| 9.27.1 Motivation: Data Compression | 243 |
| 9.27.2 Principal Component Analysis (PCA) | 244 |
| 9.28 Neural Networks | 246 |
| 9.29 Large Scale Machine Learning | 249 |
| 9.29.1 Map Reduce | 249 |
| 9.30 Mean Shift Clustering | 250 |
| 9.30.1 References | 250 |
| 9.31 Predicting things | 250 |
| 9.31.1 Input data | 251 |
| 9.31.2 Feature selection | 252 |
| 9.31.3 Algorithm selection | 252 |
| 9.31.4 In sample vs out of sample error | 252 |
| 9.31.5 Prediction study design | 253 |
| 9.31.6 Key quantities | 253 |
| 9.31.7 Receiver Operating Characteristic (ROC) curves | 254 |
| 9.31.8 Cross validation | 254 |
| 9.31.9 Decision Trees | 255 |
| 9.31.10 Bagging (“Bootstrap aggregating”) | 257 |
| 9.31.11 Random forests | 257 |
| 9.31.12 Boosting | 257 |
| 9.32 Dirichlet Distribution | 259 |
| 9.32.1 Entropy | 260 |
| 9.32.2 Interpreting α | 260 |
| 9.33 Forecasting and timeseries prediction | 261 |
| 9.34 Maximum Likelihood Estimation (MLE) | 261 |
| 9.34.1 References | 263 |
| 9.35 Expectation Maximization | 263 |
| 9.35.1 Example | 263 |
| 9.35.2 Expectation Maximization as a Generalization of K-Means | 265 |
| 9.35.3 References | 266 |
| 9.36 Maximum a posteriori (MAP) estimation | 266 |
| 9.36.1 References | 267 |
| 9.37 Markov Chain Monte Carlo (MCMC) | 267 |

| | |
|---|-----|
| 9.37.1 Motivation | 267 |
| 9.37.2 Monte Carlo methods | 267 |
| 9.37.3 MCMC | 268 |
| 9.37.4 Gibbs Sampling | 268 |
| 9.37.5 References | 269 |
| 9.38 Softmax regression | 269 |
| 9.38.1 Hierarchical Softmax | 269 |
| 9.39 Factor Analysis | 269 |
| 9.40 Discriminant Function Analysis (DFA) | 269 |
| 9.41 Non-Negative Matrix Factorization (NMF) | 270 |
| 9.41.1 Matrix factorization | 270 |
| 9.41.2 Non-Negative Matrix Factorization (NMF) | 270 |
| 9.41.3 References | 270 |
| 9.42 Spectral Clustering (Affinity-Based Clustering) | 271 |
| 9.43 Information Gain & Entropy | 271 |
| 9.43.1 Specific Conditional Entropy | 271 |
| 9.43.2 Conditional Entropy | 271 |
| 9.43.3 Information Gain | 272 |
| 9.44 CURE (Clustering Using Representatives) | 272 |
| 9.44.1 References | 272 |
| 9.45 Model Validation | 273 |
| 9.46 Other Loss Functions | 273 |
| 9.46.1 Hinge Loss (aka Max-Margin Loss) | 273 |
| 9.46.2 Cross-entropy loss | 273 |
| 9.47 Data preprocessing | 273 |
| 9.47.1 Mean subtraction | 273 |
| 9.47.2 Normalization | 274 |
| 9.47.3 References | 274 |
| 9.48 Debugging Neural Networks/Choosing hyperparameters | 274 |
| 9.49 Generalized Linear Models | 274 |
| 9.49.1 Logistic Regression | 274 |
| 9.50 References | 275 |
| 9.51 Regression | 276 |

| | | |
|-----------|---|------------|
| 9.51.1 | Evaluating model quality | 277 |
| 9.51.2 | Outliers | 277 |
| 9.51.3 | Extrapolation | 278 |
| 9.51.4 | Collinearity | 278 |
| 9.51.5 | Model Selection | 278 |
| 9.51.6 | Logistic regression | 278 |
| 9.51.7 | Generalized linear models (GLM) | 279 |
| 9.52 | Linear Mixed Models (LMM), or just “Mixed Models” or “Hierarchical Linear Models” | 280 |
| 9.53 | Model fitting vs model selection | 283 |
| 9.53.1 | Model fitting | 283 |
| 9.53.2 | Model Selection | 284 |
| 9.53.3 | References | 285 |
| 9.53.4 | Squared error of the regression line | 285 |
| 9.54 | Residuals | 287 |
| 9.54.1 | Residual (error) variation | 287 |
| 9.54.2 | Total variation | 288 |
| 9.54.3 | Cross-validation | 288 |
| 10 | NLP | 291 |
| 10.1 | Challenges | 291 |
| 10.2 | Terminology | 292 |
| 10.3 | Data preparation | 294 |
| 10.3.1 | Sentence segmentation | 294 |
| 10.3.2 | Tokenization | 294 |
| 10.3.3 | Normalization | 294 |
| 10.3.4 | Term Frequency-Inverse Document Frequency (tf-idf) Weighting | 294 |
| 10.3.5 | The Vector Space Model (VSM) | 295 |
| 10.3.6 | Normalizing vectors | 295 |
| 10.4 | Measuring similarity between text | 295 |
| 10.4.1 | Minimum edit distance | 295 |
| 10.4.2 | Jaccard coefficient | 295 |
| 10.4.3 | Euclidean Distance | 296 |
| 10.4.4 | Cosine similarity | 296 |
| 10.5 | Probabilistic Language Models | 297 |

| | |
|--|------------|
| 10.5.1 n-grams | 298 |
| 10.6 Text Classification | 299 |
| 10.6.1 Naive Bayes | 299 |
| 10.6.2 Evaluating text classification | 301 |
| 10.7 Named Entity Recognition (NER) | 302 |
| 10.8 Relation Extraction | 302 |
| 10.8.1 Ontological Relations | 303 |
| 10.8.2 Methods | 303 |
| 10.9 Sentiment Analysis | 305 |
| 10.9.1 Sentiment Lexicons | 306 |
| 10.9.2 Challenges | 306 |
| 10.10 Summarization | 306 |
| 10.10.1 The general approach | 307 |
| 11 Neural Nets | 309 |
| 11.1 Biological basis | 309 |
| 11.2 Perceptron: a simple artificial neuron | 309 |
| 11.2.1 Activation functions | 311 |
| 11.2.2 Bias | 314 |
| 11.3 Multilayered Perceptron (MLP, Feed-Forward ANN) | 314 |
| 11.3.1 Training a perceptron via “backpropagation” | 315 |
| 11.4 Backpropagation | 316 |
| 11.4.1 References | 317 |
| 11.4.2 Alternate explanation of the chain rule | 317 |
| 11.5 Choosing the network configuration | 318 |
| 11.6 Recurrent neural networks (Feed-Back ANN) | 319 |
| 11.7 Sigmoid neurons (aka logistic neurons) | 319 |
| 11.8 Cost/loss/objective/error functions | 320 |
| 11.9 RBF (neural) network | 321 |
| 11.9.1 Radial Basis Functions (RBF) | 322 |
| 11.10 Deep neural networks | 323 |
| 11.11 Sources | 324 |
| 11.12 Convolutional Neural Networks (CNN) | 324 |
| 11.12.1 Convolutions | 326 |

| | |
|---|-----|
| 11.12.2 Convolution kernels | 329 |
| 11.12.3 References | 329 |
| 11.13 Recurrent Neural Networks (RNN) | 330 |
| 11.13.1 More on RNNs | 330 |
| 11.13.2 Caveats | 333 |
| 11.13.3 References | 333 |
| 11.14 Word Embeddings | 333 |
| 11.14.1 Modular Neural Networks | 334 |
| 11.14.2 Recursive Neural Networks | 336 |
| 11.15 Nonlinear neural nets | 337 |
| 11.16 Transfer learning | 337 |
| 11.17 Recursive Neural Tensor Networks | 337 |
| 11.17.1 References | 337 |
| 11.18 Choosing the network architecture | 337 |
| 11.18.1 References | 339 |
| 11.19 Weight initialization | 340 |
| 11.19.1 References | 340 |
| 11.20 Regularization | 340 |
| 11.20.1 L2 Regularization | 341 |
| 11.20.2 L1 Regularization | 341 |
| 11.20.3 Elastic net regularization | 341 |
| 11.20.4 Max norm constraints | 341 |
| 11.20.5 Dropout | 341 |
| 11.20.6 Recommendations | 343 |
| 11.20.7 References | 343 |
| 11.21 Training | 343 |
| 11.22 Hopfield Nets | 344 |
| 11.22.1 Recurrent NNs and stability | 344 |
| 11.23 Unsupervised neural networks | 349 |
| 11.24 In high dimensions, local minima are not problems | 349 |
| 11.25 Training on adversarial examples | 350 |
| 11.26 Training neural nets tips | 350 |
| 11.27 Neural net weight initialization | 350 |

| | |
|--|------------|
| 11.28 Activation functions | 350 |
| 11.29 Loss functions | 351 |
| 11.30 LSTM networks | 351 |
| 11.30.1 Variations on LSTM | 352 |
| 11.31 References | 353 |
| 12 Probabilistic Machine Learning | 355 |
| 12.0.1 Probabilistic modeling | 355 |
| 12.0.2 Bayesian modeling | 356 |
| 12.1 Nonparametric models | 358 |
| 12.1.1 What is a nonparametric model? | 358 |
| 12.1.2 Why use a Bayesian nonparametric approach? | 360 |
| 12.2 The Dirichlet Process | 361 |
| 12.2.1 Finite Mixture Models | 361 |
| 12.2.2 Chinese Restaurant Process | 361 |
| 12.3 References | 362 |
| 12.4 Infinite Mixture Models and the Dirichlet Process | 363 |
| 12.4.1 Chinese Restaurant Process | 363 |
| 12.4.2 Polya Urn Model | 363 |
| 12.4.3 Stick-Breaking Process | 364 |
| 12.4.4 Dirichlet Process | 364 |
| 12.4.5 Gibbs Sampling | 365 |
| 12.4.6 References | 365 |
| 12.5 Parametric models vs nonparametric models | 365 |
| 12.5.1 References | 366 |
| III Artificial Intelligence | 367 |
| 13 Search | 369 |
| 13.1 Depth-First, Hill Climbing, Beam | 369 |
| 13.1.1 “British Museum” search | 369 |
| 13.1.2 Depth-First Search | 369 |
| 13.1.3 Breadth-First Search | 371 |
| 13.1.4 Hill-Climbing Search | 372 |

| | |
|---|------------|
| CONTENTS | 18 |
| 13.1.5 Beam Search | 372 |
| 13.2 Optimal, Branch, and Bound | 373 |
| 13.2.1 Branch & Bound Search and the A* algorithm | 373 |
| 13.3 Games, Minimax, and Alpha-Beta | 376 |
| 13.4 Constraints: Search, Domain Reduction | 377 |
| 13.5 Monte Carlo Tree Search | 378 |
| 13.5.1 Multi-armed bandit | 379 |
| 13.5.2 Monte Carlo | 379 |
| 13.6 References | 380 |
| 13.7 References | 381 |
| 13.8 References | 381 |
| IV Simulation | 383 |
| V Practice | 385 |
| 14 Process | 387 |
| 14.1 Data analysis approach | 387 |
| 15 Data Collection | 389 |
| 15.1 Sampling | 389 |
| 15.2 Studies | 390 |
| 15.3 References | 390 |
| 15.4 Learning: Tips | 390 |
| 15.4.1 References | 391 |
| 16 Data Visualization | 393 |
| 16.1 Bivariate charts | 393 |
| 16.2 Histograms | 393 |
| 16.3 Scatterplots | 393 |
| 16.4 References | 393 |
| 16.5 Normal distribution | 394 |

| | |
|--|------------|
| VI Appendices | 395 |
| 17 Data analysis with pandas | 397 |
| 17.1 Dealing with datetimes | 402 |
| 17.2 Loading data | 402 |
| 17.3 Plotting | 403 |
| 17.3.1 Initial setup | 403 |
| 17.3.2 Basics | 403 |
| 17.3.3 Plot a cross tab | 403 |
| 17.3.4 Plot subplots as a grid | 404 |
| 17.3.5 Plot overlays | 404 |
| 17.3.6 Other plots | 404 |
| 17.3.7 Decorating plots | 405 |
| 17.3.8 Saving a figure | 405 |
| 17.4 iPython Notebooks | 405 |

Introduction

These are my notes which are broadly intended to cover the basics necessary for data science, machine learning, and artificial intelligence. They have been collected from a variety of different sources, which I include as references when I remember to - so take this as a disclaimer that most of this material is adapted, sometimes directly copied, from elsewhere. Maybe it's better to call this a "remix" or "katamari" sampled from resources elsewhere. I have tried to give credit where it is due, but sometimes I forget to include all my references, so I will generally just say that I take no credit for any material here.

Many of the graphics and illustrations are of my own creation or have been re-created from others, but plenty have also been sourced from elsewhere - again, I have tried to give credit where it is due, but some things slip through.

Data science, machine learning, and artificial intelligence are huge fields that share some foundational overlap but go in quite different directions. These notes are not comprehensive but aim to cover a significant portion of that common ground (and a bit beyond too). These notes are intended to provide intuitive understandings rather than rigorous proofs; if you are interested in them there are many other resources which will help with that.

Since mathematical concepts typically have many different applications and interpretations and often are arrived at through different disciplines and perspectives, I try to explain these concepts in as many ways as possible.

This is still very much a work in progress and it will be changing a lot! A lot may be out of order, missing, littered with TO DOs, etc.

Part I

Foundations

1

Functions

Fundamentally, a function is a relationship (mapping) between the values of some set X and some set Y :

$$f : X \rightarrow Y$$

A function can map a set to itself. For example, $f(x) = x^2$, also notated $f : x \mapsto x^2$, is the mapping of all real numbers to all real numbers, or $f : \mathbb{R} \rightarrow \mathbb{R}$.

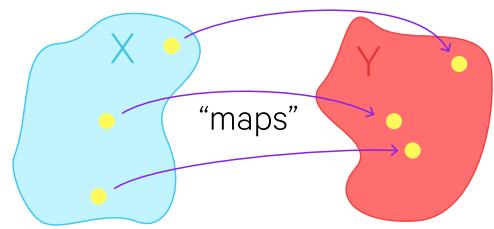
The set you are mapping *from* is called the **domain**. The set that is being mapped *to* is called the **codomain**. The **range** is the subset of the codomain which the function actually maps to (a function doesn't necessarily map to every value in the codomain. But where it does, the range = the codomain).

Functions which map to \mathbb{R} are known as **scalar-valued** or **real-valued** functions. Functions which map to \mathbb{R}^n where $n > 1$ are known as **vector-valued** functions.

1.0.1 Identity functions

An identity function maps something to itself:

$$\begin{aligned} I_X : X &\rightarrow X \\ I_X(a) &= a, \forall a \in X \end{aligned}$$



A function is a mapping between domains.

1.0.2 The inverse of a function

Say we have a function $f : X \rightarrow Y$, where $f(a) = b$ for any $a \in X$.

We say f is **invertible** if and only if there exists a function $f^{-1} : Y \rightarrow X$ such that $f^{-1} \circ f = I_X$ and $f \circ f^{-1} = I_Y$.

The inverse of a function is *unique*, that is, it is *surjective* and *injective*, that is, there is a unique x for each y .

1.0.3 Surjective functions

A **surjective** function, also called “onto”, is a function $f : X \rightarrow Y$ where, for every $y \in Y$ there exists *at least* one $x \in X$ such that $f(x) = y$. That is, every y has at least one corresponding x value.

This can also be expressed as:

$$\text{im}(f) = Y$$

since the image of the transformation encompasses the entire codomain Y . Because of that, this can also be represented as:

$$\text{range}(f) = Y$$

1.0.4 Injective functions

An **injective** function, also called “one-to-one”, is a function $f : X \rightarrow Y$ where, for every $y \in Y$, there exists *at most* one $x \in X$ such that $f(x) = y$. That is, not all y necessarily has a corresponding x , but those that do only have *one* corresponding x .

1.0.5 Surjective & injective functions

A function can be both surjective and injective, which just means that for every $y \in Y$ there exists exactly one $x \in X$ such that $f(x) = y$, that is, every y has exactly one corresponding x .

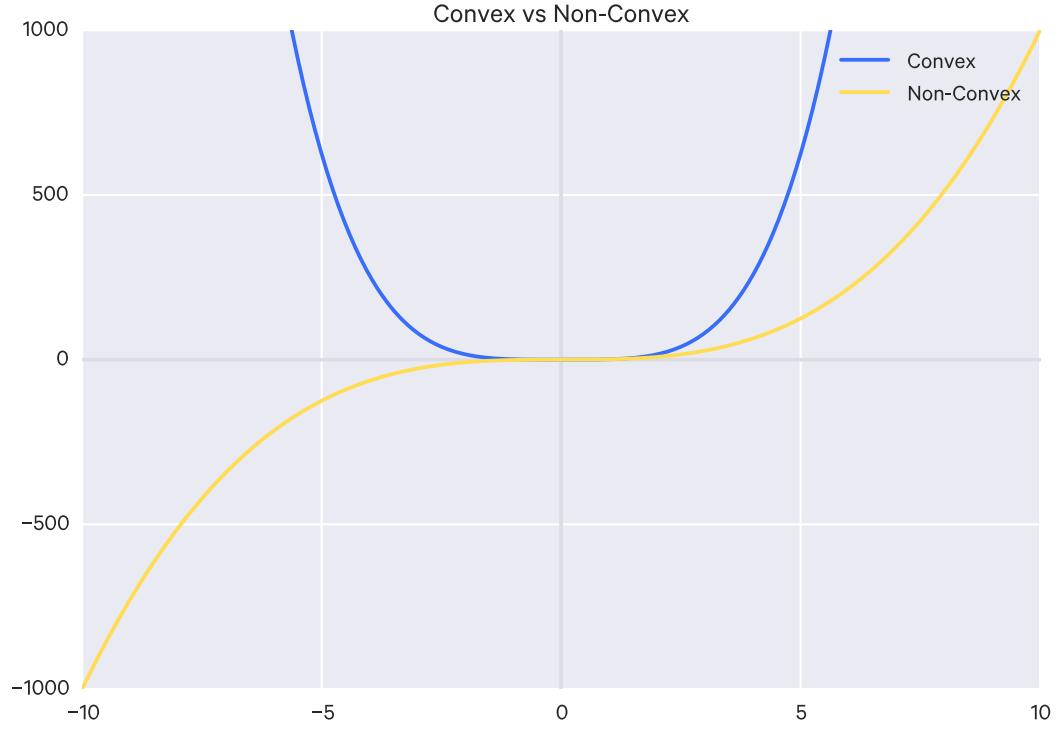
The inverse of a function is surjective and injective!

1.0.6 Convex and non-convex functions

A convex function is a continuous function whose value at the midpoint of every interval in its domain does not exceed the arithmetic mean of its values at the ends of the interval. [source](#).

A convex region is one in which any two points in the region can be joined by a straight line that does not leave the region.

Which is to say that a convex function has a minimum.



Convex and non-convex functions

1.1 References

- <http://mathworld.wolfram.com/ConvexFunction.html>

2

Linear Algebra

When working with data, we typically deal with many data points consisting of many **dimensions**. That is, each data point may have several **components**; e.g. if people are your data points, they may be represented by their height, weight, and age, which constitutes three dimensions all together. These data points of many components are called **vectors**. These are contrasted with individual values, which are called **scalars**.

We deal with these data points - vectors - simultaneously in aggregates known as **matrices**.

Linear algebra provides the tools needed to manipulate vectors and matrices.

2.1 Vectors

Vectors have **magnitude** and **direction**, e.g. 5 miles/hour going east.

Formally, this example would be represented:

$$\vec{v} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$$

since we are going 5 on x-axis and 0 on the y-axis.

Note that often the arrow is dropped, i.e. the vector is notated as just v .

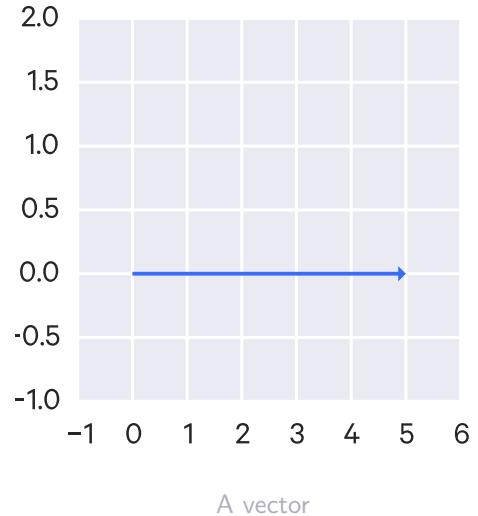
2.1.1 Real coordinate spaces

Vectors are plotted and manipulated in space. A two-dimensional vector, such as the previous example, may be represented in a two-dimensional space. A vector with three components would be represented in a three-dimensional space, and so on for any arbitrary n dimensions.

A real coordinate space (that is, a space consisting of real numbers) of n dimensions is notated \mathbb{R}^n . Such a space encapsulates all possible vectors of that dimensionality, i.e. all possible vectors of the form $[v_1, v_2, \dots, v_n]$.

To denote a vector of n dimensions, we write $x \in \mathbb{R}^n$.

For example: the notation for the two-dimensional real coordinate space is \mathbb{R}^2 , which is all possible real-valued 2-tuples (i.e. all 2D vectors whose components are real numbers). If we wanted to describe an arbitrary two-dimensional vector, we could do so with $\vec{v} \in \mathbb{R}^2$.



2.1.2 Column and row vectors

A vector $x \in \mathbb{R}^n$ typically denotes a **column vector**, i.e. with n rows and 1 column.

A **row vector** $x \in \mathbb{R}^n$ has 1 row and n columns.

2.1.3 Transposing a vector

Transposing a vector means turning its rows into columns:

$$\vec{a} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, \vec{a}^T = [x_1 \ x_2 \ x_3 \ x_4]$$

So a column vector x can be represented as a row vector with x^T .

2.1.4 Vectors operations

Vector addition

Vectors are added by adding the individual corresponding components:

$$\begin{bmatrix} 6 \\ 2 \end{bmatrix} + \begin{bmatrix} -4 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 + -4 \\ 2 + 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \end{bmatrix}$$

Multiplying a vector by a scalar

To multiply a vector with a scalar, you just multiply the individual components of the vector by the scalar:

$$3 \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \times 2 \\ 3 \times 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 3 \end{bmatrix}$$

This changes the *magnitude* of the vector, but *not the direction*.

Vector dot products

The **dot product** (also called **inner product**) of two vectors $\vec{a}, \vec{b} \in \mathbb{R}^n$ (that is, they must be of the same dimension) is denoted:

$$\vec{a} \cdot \vec{b}$$

It is calculated:

$$\vec{a} \cdot \vec{b} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{i=1}^n a_i b_i$$

Which results in a scalar value.

Note that sometimes the dot operator is dropped, so a dot product may be notated as just $\vec{a}\vec{b}$.

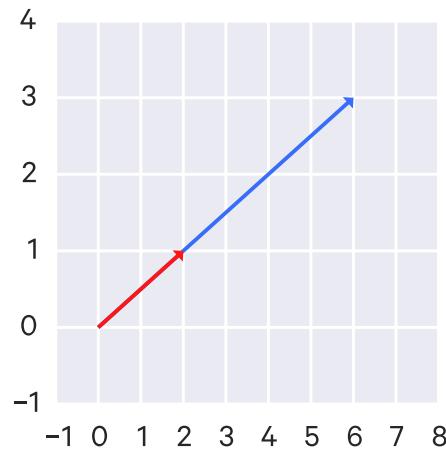
Properties of vector dot products:

- **Commutative property:** The order of the dot product doesn't matter: $\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$
- **Distributive property:** You can distribute terms in dot products: $(\vec{v} + \vec{w}) \cdot \vec{x} = (\vec{v} \cdot \vec{x} + \vec{w} \cdot \vec{x})$
- **Associative property:** $(c\vec{v}) \cdot \vec{w} = c(\vec{v} \cdot \vec{w})$

2.1.5 Norms

The **norm** of a vector $x \in \mathbb{R}^n$, denoted $\|x\|$, is the “length” of the vector.

There are many different norms, the most common of which is the Euclidean norm (also known as the ℓ_2 norm), denoted $\|x\|_2$, computed:



Example: The red vector is before multiplying a scalar, blue is after.

$$\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2} = \sqrt{x^T x}$$

Generally, a norm is just any function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ which satisfies the following properties:

1. *non-negativity*: For all $x \in \mathbb{R}^n$, $f(x) \geq 0$
2. *definiteness*: $f(x) = 0$ if and only if $x = 0$
3. *homogeneity*: For all $x \in \mathbb{R}^n$, $t \in \mathbb{R}$, $f(tx) = |t|f(x)$
4. *triangle inequality*: For all $x, y \in \mathbb{R}^n$, $f(x + y) \leq f(x) + f(y)$

Another norm is the ℓ_1 norm:

$$\|x\|_1 = \sum_{i=1}^n |x_i|$$

and the ℓ_∞ norm:

$$\|x\|_\infty = \max_i |x_i|$$

These three norms are part of the family of ℓ_p norms, which are parameterized by a real number $p \geq 1$, and defined as:

$$\|x\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$$

There are also norms for matrices.

Lengths and dot products

You may notice that the dot product of a vector with itself is the square of that vector's length:

$$\vec{a} \cdot \vec{a} = a_1^2 + a_2^2 + \cdots + a_n^2 = \|\vec{a}\|^2$$

So the length of a vector can be written:

$$\|\vec{a}\| = \sqrt{\vec{a} \cdot \vec{a}}$$

2.1.6 Unit vectors

Each dimension in a space has a **unit vector**, generally denoted with a hat, e.g. \hat{u} , which is a vector constrained to that dimension (that is, it has 0 magnitude in all other dimensions), with length 1, e.g. $\|\hat{u}\| = 1$.

Unit vectors exists for all \mathbb{R}^n .

The unit vector is also called a **normalized vector** (which is not to be confused with a *normal* vector, which is something else entirely.)

The unit vector in the same direction as some vector \vec{v} is found by computing:

$$\hat{u} = \frac{\vec{v}}{\|\vec{v}\|}$$

For instance, in \mathbb{R}^2 space, we would have two unit vectors:

$$\hat{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \hat{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

In \mathbb{R}^3 space, we would have three unit vectors:

$$\hat{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \hat{j} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \hat{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

But you can have unit vectors in any direction. Say you have a vector:

$$\vec{a} = \begin{bmatrix} 5 \\ -6 \end{bmatrix}$$

You can find a unit vector \hat{u} in the direction of this vector like so:

$$\hat{u} = \frac{\vec{a}}{\|\vec{a}\|}$$

so, with our example:

$$\hat{u} = \frac{1}{\|\vec{a}\|} \vec{a} = \frac{1}{\sqrt{61}} \begin{bmatrix} 5 \\ -6 \end{bmatrix} = \begin{bmatrix} \frac{5}{\sqrt{61}} \\ \frac{-6}{\sqrt{61}} \end{bmatrix}$$

2.1.7 Angles between vectors

Say you have two non-zero vectors, $\vec{a}, \vec{b} \in \mathbb{R}^n$.

We often denote the angle between two vectors as θ .

The **law of cosine** tells us, that for a triangle:

$$C^2 = A^2 + B^2 - 2AB \cos \theta$$

Using this law, we can get the angle between our two vectors:

$$\|\vec{a} - \vec{b}\|^2 = \|\vec{b}\|^2 + \|\vec{a}\|^2 - 2\|\vec{a}\|\|\vec{b}\| \cos \theta$$

which simplifies to:

$$\vec{a} \cdot \vec{b} = \|\vec{a}\|\|\vec{b}\| \cos \theta$$

There are two special cases if the vectors are **collinear**, that is if $\vec{a} = c\vec{b}$:

- If $c > 0$, then $\theta = 0$.
- If $c < 0$, then $\theta = 180^\circ$

2.1.8 Perpendicular vectors

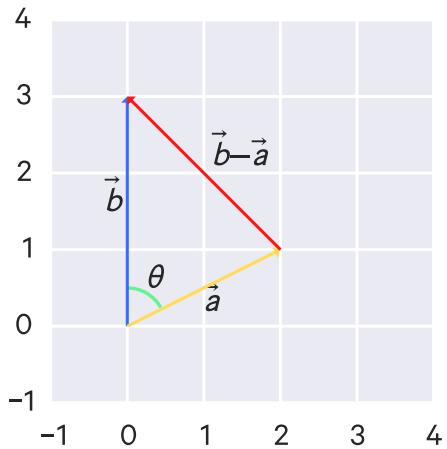
With the above angle calculation, you can see that if \vec{a} and \vec{b} are non-zero, and their dot product is 0, that is, $\vec{a} \cdot \vec{b} = \vec{0}$, then they are perpendicular to each other.

Whenever a pair of vectors satisfies this condition $\vec{a} \cdot \vec{b} = \vec{0}$, it is said that the two vectors are **orthogonal**.

Note that because any vector times the zero vector equals the zero vector: $\vec{0} \cdot \vec{x} = \vec{0}$.

Thus the zero vector is orthogonal to *everything*.

Technical detail: So if the vectors are both non-zero and orthogonal, then the vectors are *both* perpendicular and orthogonal. But of course, since the zero vector is not non-zero, it cannot be perpendicular to anything, but it is orthogonal to everything.



Angle between two vectors.

2.1.9 Normal vectors

A **normal** vector is one which is perpendicular to all the points/vectors on a plane.

That is for any vector \vec{a} on the plane, and a normal vector, \vec{n} , to that plane, we have:

$$\vec{n} \cdot \vec{a} = 0$$

For example: given an equation of a plane, $Ax + By + Cz = D$, the normal vector is simply:

$$\vec{n} = A\hat{i} + B\hat{j} + C\hat{k}$$

2.1.10 Orthonormal vectors

Given $V = \{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k\}$ where:

- $||\vec{v}_i|| = 1$ for $i = 1, 2, \dots, k$. That is, the length of each vector in V is 1 (that is, they have all been normalized).
- $\vec{v}_i \cdot \vec{v}_j = 0$ for $i \neq j$. That is, these vectors are all orthogonal to each other.

This can be summed up as:

$$\vec{v}_i \cdot \vec{v}_j = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

This is an **orthonormal** set. The term comes from the fact that these vectors are all orthogonal to each other, and they have all been normalized.

2.1.11 Additional vector operations

These vector operations are less common, but included for reference.

Vector outer products

For **outer products**, the two vectors do not need to be of the same dimension (i.e. $x \in \mathbb{R}^n, y \in \mathbb{R}^m$), and the result is a matrix instead of a scalar:

$$xy \in \mathbb{R}^{n \times m} = \begin{bmatrix} x_1y_1 & \cdots & x_1y_m \\ \vdots & \ddots & \vdots \\ x_ny_1 & \cdots & x_ny_m \end{bmatrix}$$

Vector cross products

Cross products are much more limited than dot products. Dot products can be calculated for any \mathbb{R}^n . Cross products are only defined in \mathbb{R}^3 .

Unlike the dot product, which results in a scalar, the cross product results in a vector which is orthogonal to the original vectors (i.e. it is orthogonal to the plane defined by the two original vectors).

$$\vec{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$\vec{a} \times \vec{b} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

For example:

$$\begin{bmatrix} 1 \\ -7 \\ 1 \end{bmatrix} \times \begin{bmatrix} 5 \\ 2 \\ 4 \end{bmatrix} = \begin{bmatrix} -7 \times 4 - 1 \times 2 \\ 1 \times 5 - 1 \times 4 \\ 1 \times 2 - -7 \times 5 \end{bmatrix} = \begin{bmatrix} -30 \\ 1 \\ 37 \end{bmatrix}$$

2.2 Linear Combinations

2.2.1 Parametric representations of lines

Any line in an n -dimensional space can be represented using vectors.

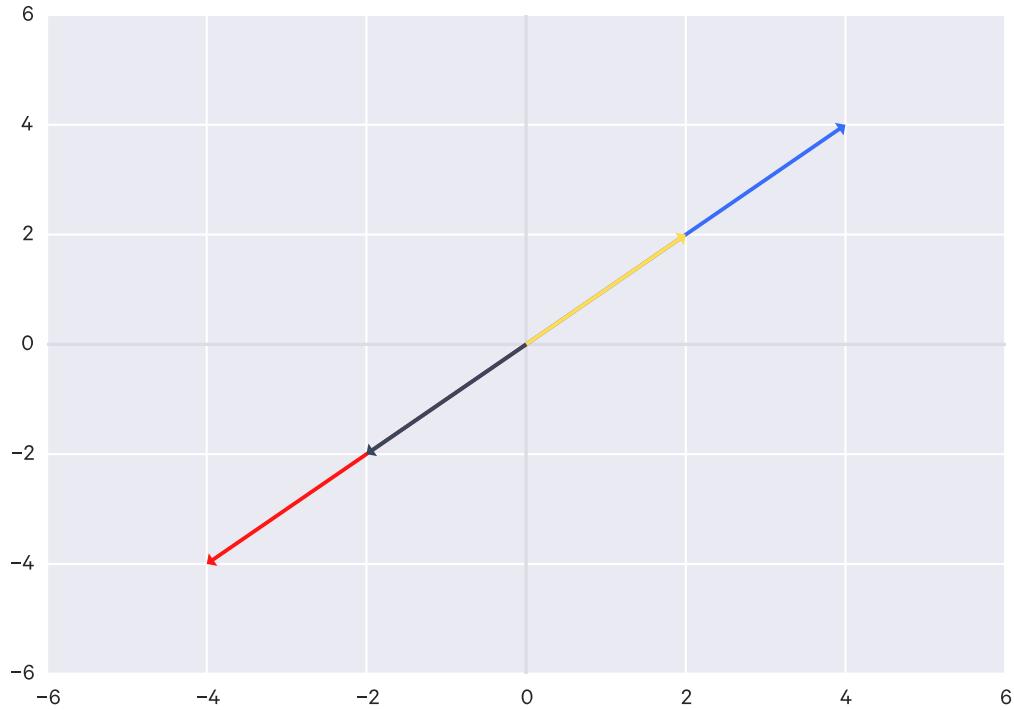
Say you have a vector \vec{v} and a set S consisting of all scalar multiplications of that vector (where the scalar c is any real number):

$$S = \{c\vec{v} \mid c \in \mathbb{R}\}$$

This set S represents a line, since multiplying a vector with scalars does not change its direction, only its magnitudes, so that set of vectors covers the entirety of the line.

But that line is around the origin. If you wanted to shift it, you need only to add a vector, which we'll call \vec{x} . So we could define a line as:

$$L = \{\vec{x} + c\vec{v} \mid c \in \mathbb{R}\}$$



For example: say you are given two vectors:

$$\vec{a} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \vec{b} = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$$

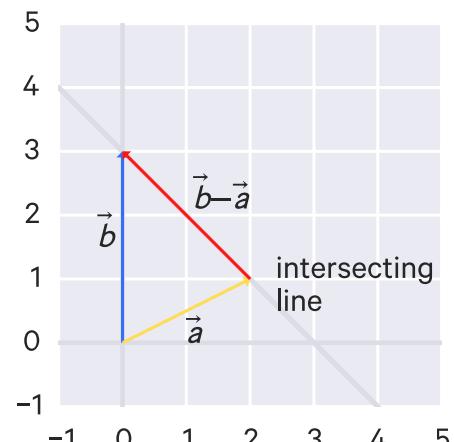
Say you want to find the line that goes through them. First you need to find the vector along that intersecting line, which is just $\vec{b} - \vec{a}$.

Although in standard form, that vector originates at the origin.

Thus you still need to shift it by finding the appropriate vector \vec{x} to add to it. But as you can probably see, we can use our \vec{a} to shift it, giving us:

$$L = \{\vec{a} + c(\vec{b} - \vec{a}) \mid c \in \mathbb{R}\}$$

And this works for any arbitrary n dimensions! (Although in other spaces, this wouldn't really be a "line". In \mathbb{R}^3 space, for instance, this would define a plane.)



You can convert this form to a **parametric** equation, where the equation for a dimension of the vector a_i looks like:

$$a_i + (b_i - a_i)c$$

Say you are in \mathbb{R}^2 so, you might have:

$$L = \left\{ \begin{bmatrix} 0 \\ 3 \end{bmatrix} + c \begin{bmatrix} -2 \\ 2 \end{bmatrix} \mid c \in \mathbb{R} \right\}$$

you can write it as the following parametric equation:

$$\begin{aligned} x &= 0 + -2c = -2c \\ y &= 3 + 2c = 2c + 3 \end{aligned}$$

2.2.2 Linear combinations

Say you have the following vectors in \mathbb{R}^m :

$$\vec{v}_1, \vec{v}_2, \dots, \vec{v}_n$$

A **linear combination** is just some sum of the combination of these vectors, scaled by arbitrary constants ($c_1 \rightarrow c_n \in \mathbb{R}$):

$$c_1 \vec{v}_1 + c_2 \vec{v}_2 + \dots + c_n \vec{v}_n$$

For example:

$$\vec{a} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \vec{b} = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$$

A linear combination would be:

$$0\vec{a} + 0\vec{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Any vector in the space \mathbb{R}^2 can be represented by some linear combination of these two vectors.

2.2.3 Spans

The set of all linear combinations for some vectors is called the **span**.

The span of some vectors can define an entire space. For instance, using our previously-defined vectors:

$$\text{span}(\vec{a}, \vec{b}) = \mathbb{R}^2$$

But this is not always true for the span of any arbitrary set of vectors. For instance, this does *not* represent all the vectors in \mathbb{R}^2 :

$$\text{span}\left(\begin{bmatrix}-2\\-2\end{bmatrix}, \begin{bmatrix}2\\2\end{bmatrix}\right)$$

These two vectors are **collinear** (that is, they lie along the same line), so combinations of them will only yield other vectors along that line.

As another example, the span of the zero vector, $\text{span}(\vec{0})$, cannot represent all vectors in a space.

Formally, the span is defined as:

$$\text{span}(\vec{v}_1, \vec{v}_2, \dots, \vec{v}_n) = \{c_1\vec{v}_1 + c_2\vec{v}_2 + \dots + c_n\vec{v}_n \mid c_i \in \mathbb{R} \forall 1 \leq i \leq n\}$$

2.2.4 Linear independence

The set of vectors in the previous collinear example:

$$\left\{\begin{bmatrix}-2\\-2\end{bmatrix}, \begin{bmatrix}2\\2\end{bmatrix}\right\}$$

are called a **linearly dependent set**, which means that some vector in the set can be represented as the linear combination of some of the other vectors in the set.

In this example, we could represent $\begin{bmatrix}-2\\-2\end{bmatrix}$ using the linear combination of the other vector, i.e. $-1\begin{bmatrix}2\\2\end{bmatrix}$.

You can think of a linearly dependent set as one that contains a redundant vector - one that doesn't add any more information to the set.

As another example:

$$\left\{\begin{bmatrix}2\\3\end{bmatrix}, \begin{bmatrix}7\\2\end{bmatrix}, \begin{bmatrix}9\\5\end{bmatrix}\right\}$$

is linearly dependent because $\vec{v}_1 + \vec{v}_2 = \vec{v}_3$.

Naturally, a set that is *not* linearly dependent is called a **linearly independent set**.

For a more formal definition of linear dependence, a set of vectors:

$$S = \{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_n\}$$

is linearly dependent iff (if and only if)

$$c_1\vec{v}_1 + c_2\vec{v}_2 + \dots + c_n\vec{v}_n = \vec{0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

for some c_i 's where *at least one is non-zero*.

To put the previous examples in context, if you can show that at least one of the vectors can be described by the linear combination of the other vectors in the set, that is:

$$\vec{v}_1 = a_2\vec{v}_2 + a_3\vec{v}_3 + \dots + a_n\vec{v}_n$$

then you have a linearly dependent set because that can be reduced to show:

$$\vec{0} = -1\vec{v}_1 + a_2\vec{v}_2 + a_3\vec{v}_3 + \dots + a_n\vec{v}_n$$

Thus you can calculate the zero vector as a linear combination of the vectors where at least one constant is non-zero, which satisfies the definition for linear dependence.

So then a set is linearly *independent* if, to calculate the zero vector as a linear combination of the vectors, the coefficients must all be zero.

Going back to spans, the span of set of size n which is *linearly independent* can describe that set's entire space (e.g. \mathbb{R}^n).

An example problem:

Say you have the set:

$$S = \left\{ \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 2 \end{bmatrix} \right\}$$

and you want to know:

- does $\text{span}(S) = \mathbb{R}^3$?
- is S linearly independent?

For the first question, you want to see if any linear combination of the set yields any arbitrary vector in \mathbb{R}^3 :

$$c_1 \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}, c_2 \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}, c_3 \begin{bmatrix} -1 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

You can distribute the coefficients:

$$\begin{bmatrix} 1c_1 \\ -1c_1 \\ 2c_1 \end{bmatrix}, \begin{bmatrix} 2c_2 \\ 1c_2 \\ 3c_2 \end{bmatrix}, \begin{bmatrix} -1c_3 \\ 0c_3 \\ 2c_3 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

So you can break that out into a system of equations:

$$\begin{aligned} c_1 + 2c_2 - c_3 &= a \\ -c_1 + c_2 + 0 &= b \\ 2c_1 + 3c_2 + 2c_3 &= c \end{aligned}$$

And solve it, which gives you:

$$\begin{aligned} c_3 &= \frac{1}{11}(3c - 5a + b) \\ c_2 &= \frac{1}{3}(b + a + c_3) \\ c_1 &= a - 2c_2 + c_3 \end{aligned}$$

So it looks like you can get these coefficients from any a, b, c , so we can say $\text{span}(S) = \mathbb{R}^3$.

For the second question, we want to see if all of the coefficients have to be non-zero for this to be true:

$$c_1 \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}, c_2 \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}, c_3 \begin{bmatrix} -1 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

We can just reuse the previous equations we derived for the coefficients, substituting $a = 0, b = 0, c = 0$, which gives us:

$$c_1 = c_2 = c_3 = 0$$

So we know this set is linearly independent.

2.3 Matrices

The notation $m \times n$ in terms of matrices mean there are m rows and n columns.

So that matrix would look like:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

A matrix of these dimensions may also be notated $\mathbb{R}^{m \times n}$ to indicate its membership in that set.

We refer to the entry in the i th row and j th column with the notation \mathbf{A}_{ij} .

2.3.1 Matrix operations

Matrix addition

Matrices must have the same dimensions in order to be added (or subtracted).

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix}$$

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \dots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2n} + b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \dots & a_{mn} + b_{mn} \end{bmatrix}$$

$$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$$

Matrix-scalar multiplication

Just distribute the scalar:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, c\mathbf{A} = \begin{bmatrix} ca_{11} & ca_{12} & \dots & ca_{1n} \\ ca_{21} & ca_{22} & \dots & ca_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ ca_{m1} & ca_{m2} & \dots & ca_{mn} \end{bmatrix}$$

Matrix-vector products

To multiply a $m \times n$ matrix with a vector, the vector must have n components (that is, the same number of components as there are columns in the matrix, i.e. $\vec{x} \in \mathbb{R}^n$):

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

The product would be:

$$\mathbf{A}\vec{x} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}$$

This results in a $m \times 1$ matrix.

Matrix-vector products as linear combinations

If you interpret each column in a matrix \mathbf{A} as its own vector \vec{v}_i , such that:

$$\mathbf{A} = [\vec{v}_1 \quad \vec{v}_2 \quad \dots \quad \vec{v}_n]$$

Then the product of a matrix and vector can be rewritten simply as a linear combination of those vectors:

$$\mathbf{A}\vec{x} = x_1\vec{v}_1 + x_2\vec{v}_2 + \cdots + x_n\vec{v}_n$$

Matrix-vector products as linear transformations

A matrix-vector product can also be seen as a linear transformation. You can describe it as a transformation:

$$\begin{aligned} \mathbf{A} &= [\vec{v}_1 \quad \vec{v}_2 \quad \dots \quad \vec{v}_n] \\ T : \mathbb{R}^n &\rightarrow \mathbb{R}^m \\ T(\vec{x}) &= \mathbf{A}\vec{x} \end{aligned}$$

It satisfies the conditions for a linear transformation (not shown here), so a matrix-vector product is always a linear transformation.

Just to be clear: the transformation of a vector can always be expressed as that vector's product with some matrix; that matrix is referred to as the **transformation matrix**.

So in the equations above, \mathbf{A} is the transformation matrix.

To reiterate:

- any matrix-vector product is a linear transformation
- any linear transformation can be expressed in terms of a matrix-vector product

Matrix-matrix products

To multiply two matrices, one must have the same number of columns as the other has rows. That is, you can only multiply an $m \times n$ matrix with an $n \times p$ matrix. The resulting matrix will be of $m \times p$ dimensions.

That is, if $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, then $C = AB \in \mathbb{R}^{m \times p}$.

The resulting matrix is defined as such:

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$$

You can break the terms out into individual matrix-vector products. Then you combine the resulting vectors to get the final matrix.

More formally, the i th column of the resulting product matrix is obtained by multiplying \mathbf{A} with the i th column of \mathbf{B} for $i = 1, 2, \dots, k$.

$$\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 3 \\ 0 & 1 \\ 5 & 2 \end{bmatrix}$$

The product would be:

$$\begin{aligned} \begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix} &= \begin{bmatrix} 11 \\ 9 \end{bmatrix} \\ \begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix} &= \begin{bmatrix} 10 \\ 14 \end{bmatrix} \\ \begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 3 \\ 0 & 1 \\ 5 & 2 \end{bmatrix} &= \begin{bmatrix} 11 & 10 \\ 9 & 14 \end{bmatrix} \end{aligned}$$

Properties of matrix multiplication

Matrix multiplication is *not commutative*. That is, for matrices \mathbf{A} and \mathbf{B} , in general $\mathbf{A} \times \mathbf{B} \neq \mathbf{B} \times \mathbf{A}$. They may not even be of the same dimension.

Matrix multiplication is *associative*. For example, for matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, we can say that:

$$\mathbf{A} \times \mathbf{B} \times \mathbf{C} = \mathbf{A} \times (\mathbf{B} \times \mathbf{C}) = (\mathbf{A} \times \mathbf{B}) \times \mathbf{C}$$

There is also an *identity matrix* \mathbf{I} . For any matrix \mathbf{A} , we can say that:

$$\mathbf{A} \times \mathbf{I} = \mathbf{I} \times \mathbf{A} = \mathbf{A}$$

2.3.2 The identity matrix

The **identity matrix** is an $n \times n$ matrix where every component is 0, except for those along the diagonal:

$$\mathbf{I}_n = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

When you multiply the identity matrix by any vector:

$$\mathbf{I}_n \vec{x} \mid \vec{x} \in \mathbb{R}^n = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \vec{x}$$

That is, a vector multiplied by the identity matrix equals itself.

2.3.3 Diagonal matrices

A **diagonal matrix** is a matrix where all non-diagonal elements are 0, typically denoted $\text{diag}(x_1, x_2, \dots, x_n)$, where

$$D_{ij} = \begin{cases} d_i & i = j \\ 0 & i \neq j \end{cases}$$

So the identity matrix is $I = \text{diag}(1, 1, \dots, 1)$.

2.3.4 Some properties of matrices

Associative property

$$(AB)C = A(BC)$$

i.e. it doesn't matter where the parentheses are.

This applies to compositions as well:

$$(h \circ f) \circ g = h \circ (f \circ g)$$

Distributive property

$$A(B + C) = AB + AC$$

$$(B + C)A = BA + CA$$

2.3.5 Matrix inverses

If \mathbf{A} is an $m \times m$ matrix, and if it has an inverse, then:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

Only square matrices can have inverses. An inverse does not exist for all square matrices, but those that have one are called **invertible** or **non-singular**, otherwise they are **non-invertible** or **singular**.

The inverse exists if and only if A is full rank.

The invertible matrices $A, B \in \mathbb{R}^{n \times n}$ have the following properties:

- $(A^{-1})^{-1} = A$
- If $Ax = B$, we can multiply by A^{-1} on both sides to obtain $x = A^{-1}b$
- $(AB)^{-1} = B^{-1}A^{-1}$
- $(A^{-1})^T = (A^T)^{-1}$; this matrix is often denoted A^{-T}

2.3.6 Matrix determinants

The determinant of a square matrix $A \in \mathbb{R}^{n \times n}$ is a function $\det : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}$, denoted $|A|$, $\det(A)$, or sometimes with the parentheses dropped, $\det A$.

Inverse and determinant for a 2×2 matrix

Say you have the matrix:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

You can calculate the inverse of this matrix as:

$$\mathbf{A}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Note that \mathbf{A}^{-1} is undefined if $ad - bc = 0$, which means that \mathbf{A} is not invertible.

The denominator $ad - bc$ is called the **determinant**. It is notated as:

$$\det(\mathbf{A}) = |\mathbf{A}| = ad - db$$

Inverse and determinant for an $n \times n$ matrix

Say we have an $n \times n$ matrix \mathbf{A} .

A submatrix of \mathbf{A}_{ij} is an $(n-1) \times (n-1)$ matrix constructed from \mathbf{A} by ignoring the i^{th} row and the j^{th} column of \mathbf{A} , which we denote by $\mathbf{A}_{\neg i, \neg j}$.

You can calculate the determinant of an $n \times n$ matrix \mathbf{A} by using some i^{th} row of \mathbf{A} , where $1 \leq i \leq n$:

$$\det(\mathbf{A}) = \sum_{j=1}^n (-1)^{i+j} a_{ij} \det(\mathbf{A}_{\neg i, \neg j})$$

All the $\det(\mathbf{A}_{ij})$ eventually reduce to the determinant of a 2×2 matrix.

Scalar multiplication of determinants

For an $n \times n$ matrix \mathbf{A} ,

$$\det(k\mathbf{A}) = k^n \det(\mathbf{A})$$

Properties of determinants

- For $A \in \mathbb{R}^{n \times n}$, $t \in \mathbb{R}$, multiplying a single row by the scalar t yields a new matrix B , for which $|B| = t|A|$.
- For $A \in \mathbb{R}^{n \times n}$, $|A| = |A^T|$
- For $A, B \in \mathbb{R}^{n \times n}$, $|AB| = |A||B|$
- For $A, B \in \mathbb{R}^{n \times n}$, $|A| = 0$ if A is singular (i.e. non-invertible).
- For $A, B \in \mathbb{R}^{n \times n}$, $|A|^{-1} = \frac{1}{|A|}$ if A is non-singular (i.e. invertible).

2.3.7 Transpose of a matrix

The **transpose** of a matrix \mathbf{A} is that matrix with its columns and rows swapped, denoted \mathbf{A}^T .

More formally, let \mathbf{A} be an $m \times n$ matrix, and let $\mathbf{B} = \mathbf{A}^T$. Then \mathbf{B} is an $n \times m$ matrix, and $B_{ij} = A_{ji}$.

- Transpose of determinants:** The determinant of a transpose is the same as the determinant of the original matrix: $\det(\mathbf{A}^T) = \det(\mathbf{A})$
- Transposes of sums:** With matrices A, B, C where $C = A + B$, then $C^T = (A + B)^T = A^T + B^T$
- Transposes of inverses:** The transpose of the inverse is equal to the inverse of the transpose: $(A^{-1})^T = (A^T)^{-1}$
- Transposes of multiplication:** $(AB)^T = B^T A^T$
- Transpose of a vector:** for two column vectors \vec{a}, \vec{b} , we know that $\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a} = \vec{a}^T \vec{b}$, from which we can derive: $(\mathbf{A}\vec{x}) \cdot \vec{y} = \vec{x} \cdot (\mathbf{A}^T \vec{y})$ (proof omitted).

2.3.8 Symmetric matrices

A square matrix $A \in \mathbb{R}^{n \times n}$ is **symmetric** if $A = A^T$.

It is **anti-symmetric** if $A = -A^T$.

For any square matrix $A \in \mathbb{R}^{n \times n}$, the matrix $A + A^T$ is symmetric and the matrix $A - A^T$ is anti-symmetric. Thus any such A can be represented as a sum of a symmetric and an anti-symmetric matrix:

$$A = \frac{1}{2}(A + A^T) + \frac{1}{2}(A - A^T)$$

Symmetric matrices have many nice properties.

The set of all symmetric matrices of dimension n is often denoted as \mathbb{S}^n , so you can denote a symmetric $n \times n$ matrix A as $A \in \mathbb{S}^n$.

The quadratic form

Given a square matrix $A \in \mathbb{R}^{n \times n}$ and a vector $x \in \mathbb{R}^n$, the scalar value $x^T A x$ is called a **quadratic form**:

$$x^T A x = \sum_{i=1}^n \sum_{j=1}^n A_{ij} x_i x_j$$

Here A is typically assumed to be symmetric.

Types of symmetric matrices

Given a symmetric matrix $A \in \mathbb{S}^n$...

- A is **positive definite** (PD) if for all non-zero vectors $x \in \mathbb{R}^n$, $x^T A x > 0$.
 - This is often denoted $A \succ 0$ or $A > 0$.
 - The set of all positive definite matrices is denoted \mathbb{S}_{++}^n .
- A is **positive semidefinite** (PSD) if for all vectors $x \in \mathbb{R}^n$, $x^T A x \geq 0$.
 - This is often denoted $A \succeq 0$ or $A \geq 0$.
 - The set of all positive semidefinite matrices is denoted \mathbb{S}_+^n .
- A is **negative definite** (ND) if for all non-zero vectors $x \in \mathbb{R}^n$, $x^T A x < 0$.
 - This is often denoted $A \prec 0$ or $A < 0$.
- A is **negative semidefinite** (NSD) if for all vectors $x \in \mathbb{R}^n$, $x^T A x \leq 0$.
 - This is often denoted $A \preceq 0$ or $A \leq 0$.
- A is **indefinite** if it is neither positive semidefinite nor negative semidefinite, that is, if there exists $x_1, x_2 \in \mathbb{R}^n$ such that $x_1^T A x_1 > 0$ and $x_2^T A x_2 < 0$.

Some other properties of note:

- If A is positive definite, then $-A$ is negative definite and vice versa.
- If A is positive semidefinite, then $-A$ is negative semidefinite and vice versa.
- If A is indefinite, then $-A$ is also indefinite and vice versa.
- Positive definite and negative definite matrices are always invertible.
- For any matrix $A \in \mathbb{R}^{m \times n}$, which does not need to be symmetric or square, the matrix $G = A^T A$, called a **Gram matrix**, is always positive semidefinite.
 - If $m \geq n$ and A is full rank, then G is positive definite.

2.3.9 The Trace

The **trace** of a square matrix $A \in \mathbb{R}^{n \times n}$ is denoted $\text{tr}(A)$ and is the sum of the diagonal elements in the matrix:

$$\text{tr}(A) = \sum_{i=1}^n A_{ii}$$

The trace has the following properties:

- $\text{tr}(A) = \text{tr}(A^T)$
- For $B \in \mathbb{R}^{n \times n}$, $\text{tr}(A + B) = \text{tr}(A) + \text{tr}(B)$
- For $t \in \mathbb{R}$, $\text{tr}(tA) = t \text{tr}(A)$
- If AB is square, then $\text{tr}(AB) = \text{tr}(BA)$
- If ABC is square, then $\text{tr}(ABC) = \text{tr}(BCA) = \text{tr}(CAB)$ and so on for the product of more matrices

2.3.10 Orthogonal matrix

Say we have a $n \times k$ matrix \mathbf{C} , whose column rows form an orthonormal set.

If $k = n$ then \mathbf{C} is a square matrix ($n \times n$) and since \mathbf{C} 's columns are linearly independent, \mathbf{C} is invertible.

For an orthonormal matrix:

$$\begin{aligned} \mathbf{C}^T \mathbf{C} &= \mathbf{I}_n & \therefore \mathbf{C}^T &= \mathbf{C}^{-1} \\ \mathbf{C}^{-1} \mathbf{C} &= \mathbf{I}_n \end{aligned}$$

When \mathbf{C} is an $n \times n$ matrix (i.e. square) whose columns form an orthonormal set, we say that \mathbf{C} is an *orthogonal matrix*.

Orthogonal matrices have the property of $C^T C = I = CC^T$.

Orthogonal matrices also have the property that operating on a vector with an orthogonal matrix will not change its Euclidean norm, i.e. $\|Cx\|_2 = \|x\|_2$ for any $x \in \mathbb{R}^n$.

Orthogonal matrices preserve angles and lengths

For an orthogonal matrix \mathbf{C} , when you multiply \mathbf{C} by some vector, the length and angle of the vector is preserved:

$$\begin{aligned} \|\vec{x}\| &= \|\mathbf{C}\vec{x}\| \\ \cos \theta &= \cos \theta_C \end{aligned}$$

2.3.11 Adjoints

The **classical adjoint**, often just called the **adjoint** of a matrix $A \in \mathbb{R}^{n \times n}$ is denoted $\text{adj}(A)$ and defined as:

$$\text{adj}(A)_{ij} = (-1)^{i+j} |A_{\neg j, \neg i}|$$

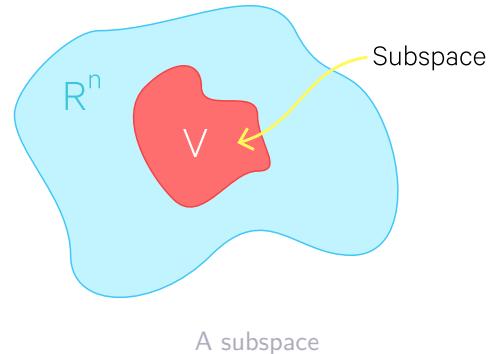
Note that the indices are switched in $A_{\neg j, \neg i}$.

2.4 Subspaces

Say we have set of vectors V which is a subset of \mathbb{R}^n , that is, every vector in the set has n components.

V is a *linear subspace* of \mathbb{R}^n if

- V contains the zero vector $\vec{0}$
- for a vector \vec{x} in V , $c\vec{x}$ (where $c \in \mathbb{R}$) must also be in V , i.e. *closure under scalar multiplication*.
- for a vector \vec{a} in V and a vector \vec{b} in V , $\vec{a} + \vec{b}$ must also be in V , i.e. *closure under addition*.



Example:

Say we have the set of vectors:

$$S = \left\{ \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2 : x_1 \geq 0 \right\}$$

which is the shaded area below.

Is S a subspace of \mathbb{R}^2 ?

- It does contain the zero vector
- It is closed under addition:

$$\begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} a+c \\ b+d \end{bmatrix}$$

Since a & b are both > 0 (that was a criteria for the set), the $a+b$ will also be greater than 0, so it will also be in the set (there were no constraints on the second component so it doesn't matter what that is)

- It is NOT closed under multiplication:

$$-1 \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} -a \\ -b \end{bmatrix}$$

Since a is ≥ 0 , $-a$ will be ≤ 0 , which falls outside the constraints of the set and thus is not contained within the set.

So no, this set is not a subspace of \mathbb{R}^2 .

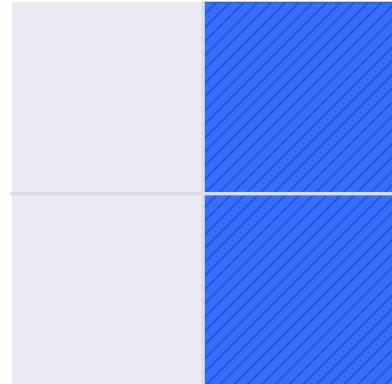
2.4.1 Spans and subspaces

Let's say we have the set:

$$U = \text{span}(\vec{v}_1, \vec{v}_2, \vec{v}_3)$$

where each vector has n components. Is this a valid subspace of \mathbb{R}^n ?

Since the span represents all the linear combinations of those vectors, we can define an arbitrary vector in the set as:



$$\vec{x} = c_1 \vec{v}_1 + c_2 \vec{v}_2 + c_3 \vec{v}_3$$

- the set does contain the zero vector:

$$0\vec{v}_1 + 0\vec{v}_2 + 0\vec{v}_3 = \vec{0}$$

- it is closed under multiplication, since the following is just another linear combination:

$$a\vec{x} = ac_1 \vec{v}_1 + ac_2 \vec{v}_2 + ac_3 \vec{v}_3$$

- it is closed under addition, since if we take another arbitrary vector in the set:

$$\vec{y} = d_1 \vec{v}_1 + d_2 \vec{v}_2 + d_3 \vec{v}_3$$

and add them:

$$\vec{x} + \vec{y} = (c_1 + d_1)\vec{v}_1 + (c_2 + d_2)\vec{v}_2 + (c_3 + d_3)\vec{v}_3$$

that's also just another linear combination in the set.

2.4.2 Basis of a subspace

If we have a subspace $V = \text{span}(S)$ where the set of vectors $S = \vec{v}_1, \vec{v}_2, \dots, \vec{v}_n$ is linearly independent, then we can say that S is a *basis* for V .

A set S is the basis for a subspace V if S is linearly independent and its span defines V . In other words, the basis is the *minimum* set of vectors that spans the subspace that it is a basis of.

All bases for a subspace will have the same number of elements.

2.4.3 Dimension of a subspace

The *dimension* of a subspace is the number of elements in a basis for that subspace.

2.4.4 Nullspace of a matrix

Say we have:

$$\mathbf{A}\vec{x} = \vec{0}$$

If you have a set N of all $x \in \mathbb{R}^n$ that satisfies this equation, do you have a valid subspace?

Of course if $\vec{x} = \vec{0}$ this equation is satisfied. So we know the zero vector is part of this set (which is a requirement for a valid subspace).

The other two properties (closure under multiplication and addition) necessary for a subspace also hold:

$$\begin{aligned} A(\vec{v}_1 + \vec{v}_2) &= A\vec{v}_1 + A\vec{v}_2 = \vec{0} \\ A(c\vec{v}_1) &= \vec{0} \end{aligned}$$

and of course $\vec{0}$ is in the set N .

So yes, the set N is a valid subspace, and it is a special subspace: the **nullspace** of \mathbf{A} , notated:

$$N(\mathbf{A})$$

That is, the nullspace for a matrix \mathbf{A} is the subspace described by the set of vectors which yields the zero vector when multiplied by \mathbf{A} , that is, the set of vectors which are the solutions for \vec{x} in:

$$\mathbf{A}\vec{x} = \vec{0}$$

Or, more formally, if \mathbf{A} is an $m \times n$ matrix:

$$N(\mathbf{A}) = \{\vec{x} \in \mathbb{R}^n \mid \mathbf{A}\vec{x} = \vec{0}\}$$

The nullspace for a matrix A may be notated $N(A)$.

Nullspace and linear independence

If you take each column in a matrix \mathbf{A} as a vector \vec{v}_i , that set of vectors is linearly independent if the nullspace of \mathbf{A} consists of *only* the zero vector. That is, if:

$$N(\mathbf{A}) = \{\vec{0}\}$$

The intuition behind this is because, if the linear combination of a set of vectors can only equal the zero vector if all of its coefficients are zero (that is, its coefficients are components of the zero vector), then it is linearly independent:

$$x_1\vec{v}_1 + x_2\vec{v}_2 + \cdots + x_n\vec{v}_n = \vec{0} \text{ iff } \vec{x} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Nullity

The **nullity** of a nullspace is its dimension, that is, it is the number of elements in a basis for that nullspace.

$$\dim(N(\mathbf{A})) = \text{nullity}(N(\mathbf{A}))$$

Left nullspace

The **left nullspace** of a matrix \mathbf{A} is the nullspace of its transpose, that is $N(\mathbf{A}^T)$:

$$N(\mathbf{A}^T) = \{\vec{x} \mid \vec{x}^T \mathbf{A} = \vec{0}^T\}$$

2.4.5 Columnspace

Again, a matrix can be represented as a set of column vectors. The **columnspace** of a matrix (also called the **range** of the matrix) is all the linear combinations (i.e. the span) of these column vectors:

$$\begin{aligned}\mathbf{A} &= [\vec{v}_1 \quad \vec{v}_2 \quad \dots \quad \vec{v}_n] \\ C(\mathbf{A}) &= \text{span}(\vec{v}_1, \vec{v}_2, \dots, \vec{v}_n)\end{aligned}$$

Because any span is a valid subspace, the columnspace of a matrix is a valid subspace.

So if you expand out the matrix-vector product, you'll see that every matrix-vector product is within that matrix's columnspace:

$$\begin{aligned}\{\mathbf{A}\vec{x} \mid \vec{x} \in \mathbb{R}^n\} \\ \mathbf{A}\vec{x} = x_1\vec{v}_1 + x_2\vec{v}_2 + \dots + x_n\vec{v}_n \\ \mathbf{A}\vec{x} = C(\mathbf{A})\end{aligned}$$

That is, for any vector in the space \mathbb{R}^n , multiplying the matrix by it just yields another linear combination of that matrix's column vectors. Therefore it is also in the columnspace.

The columnspace (range) for a matrix A may be notated $\mathcal{R}(A)$.

Rank of a columnspace

The **column rank** of a columnspace is its dimension, that is, it is the number of elements in a basis for that columnspace (i.e. the largest number of columns of the matrix which constitute a linearly independent set):

$$\dim(C(\mathbf{A})) = \text{rank}(C(\mathbf{A}))$$

Rowspace

The **rowspace** of a matrix \mathbf{A} is the columnspace of \mathbf{A}^T , i.e. $C(\mathbf{A}^T)$.

The **row rank** of a matrix is similarly the number of elements in a basis for that rowspace.

2.4.6 Rank

Note that for any matrix A , the column rank and the row rank are equal, so they are typically just referred to as $\text{rank}(A)$.

The rank has some properties:

- For $A \in \mathbb{R}^{m \times n}$, $\text{rank}(A) \leq \min(m, n)$. If $\text{rank}(A) = \min(m, n)$, then A is said to be **full rank**.
- For $A \in \mathbb{R}^{m \times n}$, $\text{rank}(A) = \text{rank}(A^T)$
- For $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$
- For $A, B \in \mathbb{R}^{m \times n}$, $\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B)$

2.4.7 The standard basis

The set of column vectors in an identity matrix I_n is known as the **standard basis for \mathbb{R}^n** .

Each of those column vectors is notated \vec{e}_i . E.g., in an identity matrix, the column vector:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \vec{e}_1$$

For a transformation $T(\vec{x})$, its transformation matrix \mathbf{A} can be expressed as:

$$\mathbf{A} = [T(\vec{e}_1) \ T(\vec{e}_2) \ \dots \ T(\vec{e}_n)]$$

2.4.8 Orthogonal compliments

Given that V is some subspace of \mathbb{R}^n , the **orthogonal compliment of V** , notated V^\perp :

$$V^\perp = \{\vec{x} \in \mathbb{R}^n \mid \vec{x} \cdot \vec{v} = 0 \ \forall \vec{v} \in V\}$$

That is, the orthogonal compliment of a subspace V is the set of all vectors where the dot product of each vector with each vector from V is 0, that is where all vectors in the set are orthogonal to all vectors in V .

V^\perp is a subspace (proof omitted).

Columnspaces, nullspaces, and transposes

$C(\mathbf{A})$ is the orthogonal compliment to $N(\mathbf{A}^T)$, and vice versa:

$$\begin{aligned} N(\mathbf{A}^T) &= C(\mathbf{A})^\perp \\ N(\mathbf{A}^T)^\perp &= C(\mathbf{A}) \end{aligned}$$

$C(\mathbf{A}^T)$ is the orthogonal compliment to $N(\mathbf{A})$, and vice versa.

$$\begin{aligned} N(\mathbf{A}) &= C(\mathbf{A}^T)^\perp \\ N(\mathbf{A})^\perp &= C(\mathbf{A}^T) \end{aligned}$$

As a reminder, columnspaces and nullspaces are spans, i.e. sets of linear combinations, i.e. lines, so these lines are orthogonal to each other.

Dimensionality and orthogonal compliments

For V , a subspace of \mathbb{R}^n :

$$\dim(V) + \dim(V^\perp) = n$$

(proof omitted here)

The intersection of orthogonal compliments

Since the vectors between a subspace and its orthogonal compliment are all orthogonal:

$$V \cap V^\perp = \{\vec{0}\}$$

That is, the only vector which exists both in a subspace and its orthogonal compliment is the zero vector.

2.4.9 Coordinates with respect to a basis

With a subspace V of \mathbb{R}^n , we have V 's basis, B , as

$$B = \{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k\}$$

We can describe any vector $\vec{a} \in V$ as a linear combination of the vectors in its basis B :

$$\vec{a} = c_1 \vec{v}_1 + c_2 \vec{v}_2 + \cdots + c_k \vec{v}_k$$

We can take these coefficients c_1, c_2, \dots, c_k as **the coordinates of \vec{a} with respect to B** , notated as:

$$[\vec{a}]_B = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix}$$

Basically what has happened here is a new coordinate system based off of the basis B is being used.

Example

Say we have $\vec{v}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $\vec{v}_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, where $B = \{\vec{v}_1, \vec{v}_2\}$ is the basis for \mathbb{R}^2 .

The point $(8, 7)$ in \mathbb{R}^2 is equal to $3\vec{v}_1 + 2\vec{v}_2$. If we set:

$$\vec{a} = 3\vec{v}_1 + 2\vec{v}_2$$

Then we can describe \vec{a} with respect to B :

$$[\vec{a}]_B = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

which looks like:

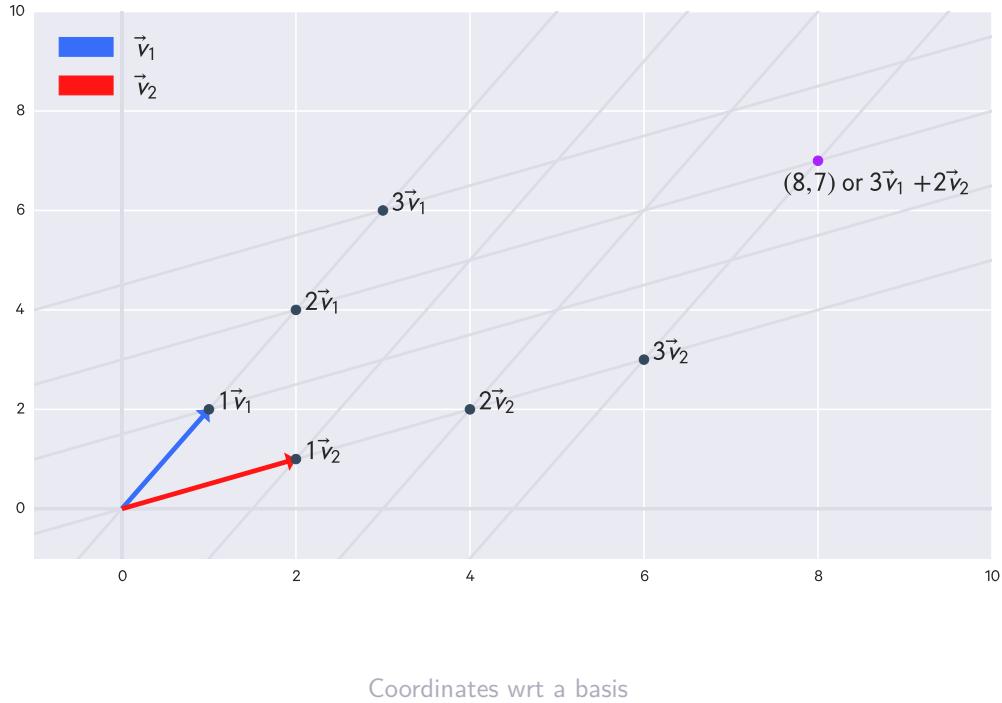
Change of basis matrix

Given the basis:

$$B = \{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k\}$$

and:

$$[\vec{a}]_B = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix}$$



say there is some $n \times k$ matrix where the column vectors are the basis vectors:

$$\mathbf{C} = [\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k]$$

We can do:

$$\mathbf{C}[\vec{a}]_B = \vec{a}$$

The matrix \mathbf{C} is known as the *change of basis matrix* and allows us to get \vec{a} in standard coordinates.

Invertible change of basis matrix

Given the basis of some subspace:

$$B = \{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k\}$$

where $\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k \in \mathbb{R}^n$, and we have a change of basis matrix:

$$\mathbf{C} = [\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k]$$

Assume:

- \mathbf{C} is invertible
- \mathbf{C} is square (that is, $k = n$, which implies that we have n basis vectors, that is, B is a basis for \mathbb{R}^n)
- \mathbf{C} 's columns are linearly independent (which they are because it is formed out of basis vectors, which by definition are linearly independent)

Under these assumptions:

- If \mathbf{C} is invertible, the span of B is equal to \mathbb{R}^n .
- If the span of B is equal to \mathbb{R}^n , \mathbf{C} is invertible.

Thus:

$$[\vec{a}]_B = \mathbf{C}^{-1} \vec{a}$$

Transformation matrix with respect to a basis

Say we have a linear transformation $T : \mathbb{R}^n \rightarrow \mathbb{R}^n$, which we can express as $T(\vec{x}) = \mathbf{A}\vec{x}$. This is with respect to the standard basis; we can say \mathbf{A} is the transformation for T with respect to the standard basis.

Say we have another basis $B = \{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_n\}$ for \mathbb{R}^n , that is, it is a basis for \mathbb{R}^n .

We could write:

$$[T(\vec{x})]_B = \mathbf{D}[\vec{x}]_B$$

and we call \mathbf{D} the transformation matrix for T with respect to the basis B .

Then we have (proof omitted):

$$\mathbf{D} = \mathbf{C}^{-1} \mathbf{A} \mathbf{C}$$

where:

- \mathbf{D} is the transformation matrix for T with respect to the basis B
- \mathbf{A} is the transformation matrix for T with respect to the standard basis
- \mathbf{C} is the change of basis matrix for B

2.4.10 Orthonormal bases

If B is an orthonormal set, it is linearly independent, and thus it could be a basis. If B is a basis, then it is an **orthonormal basis**.

Coordinates with respect to orthonormal bases

Orthonormal bases make good coordinate systems - it is much easier to find $[\vec{x}]_B$ if B is an orthonormal basis. It is just:

$$[\vec{x}]_B = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix} = \begin{bmatrix} \vec{v}_1 \cdot \vec{x} \\ \vec{v}_2 \cdot \vec{x} \\ \vdots \\ \vec{v}_k \cdot \vec{x} \end{bmatrix}$$

Note that the standard basis for \mathbb{R}^n is an orthonormal basis.

2.5 Transformations

A **transformation** is just a function which operates on vectors, which, instead of using f , is usually denoted T .

2.5.1 Linear transformations

A **linear transformation** is a transformation

$$T : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

where we can take two vectors $\vec{a}, \vec{b} \in \mathbb{R}^n$ and the following conditions are satisfied:

$$\begin{aligned} T(\vec{a} + \vec{b}) &= T(\vec{a}) + T(\vec{b}) \\ T(c\vec{a}) &= cT(\vec{a}) \end{aligned}$$

Linear transformation examples

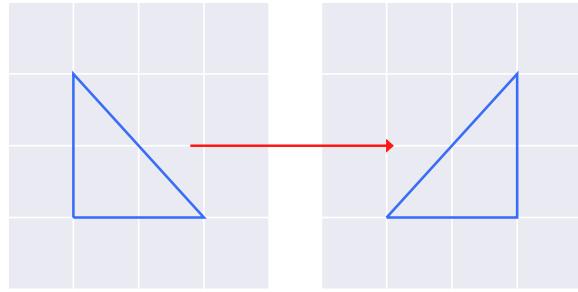
These examples are all in \mathbb{R}^2 since it's easier to visualize. But you can scale them up to any \mathbb{R}^n .

Reflection

To get from the triangle on the left and reflect it over the y -axis to get the triangle on the right, all you're doing is changing the sign of all the x values.

So a transformation would look like:

$$T\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} -x \\ y \end{bmatrix}$$



Scaling

Say you want to double the size of the triangle instead of flipping it. You'd just scale up all of its values:

An example of reflection.

$$T\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

Compositions of linear transformation

The composition of linear transformations $S(\vec{x}) = A\vec{x}$ and $T(\vec{x}) = B\vec{x}$ is denoted:

$$T \circ S(\vec{x}) = T(S(\vec{x}))$$

This is read: “the composition of T with S ”.

If $T : Y \rightarrow Z$ and $S : X \rightarrow Y$, then $T \circ S : X \rightarrow Z$.

A composition of linear transformations is also a linear transformation (proof omitted here). Because of this, this composition can also be expressed:

$$T \circ S(\vec{x}) = C\vec{x}$$

Where $C = BA$ (proof omitted), so:

$$T \circ S(\vec{x}) = BA\vec{x}$$

2.5.2 Kernels

The **kernel** of T , denoted $\ker(T)$, is all of the vectors in the domain such that the transformation of those vectors is equal to the zero vector:

$$\ker(T) = \{\vec{x} \in \mathbb{R}^n \mid T(\vec{x}) = \{\vec{0}\}\}$$

You may notice that, because $T(\vec{x}) = \mathbf{A}\vec{x}$,

$$\ker(T) = N(\mathbf{A})$$

That is, the kernel of the transformation is the same as the nullspace of the transformation matrix.

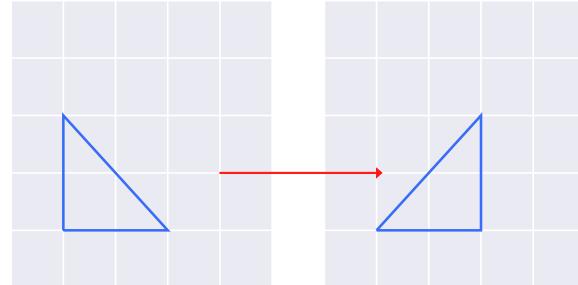
2.6 Images

2.6.1 Image of a subset of a domain

When you pass a set of vectors (i.e. a subset of a domain \mathbb{R}^n) through a transformation, the result is called the **image of the set under the transformation**. E.g. $T(S)$ is the image of S under T .

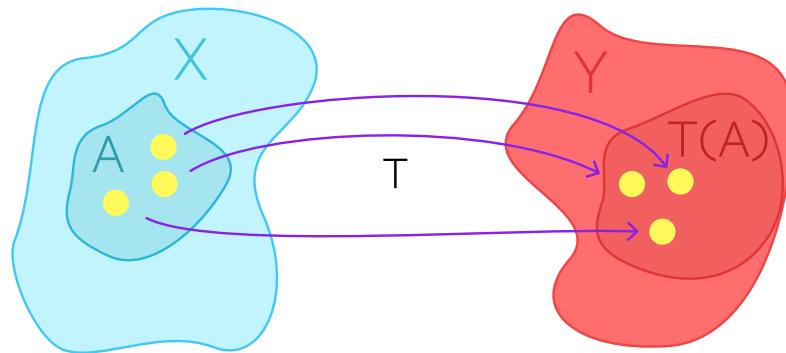
For example, say we have some vectors which define the triangle on the left. When a transformation is applied to that set, the image is the result on the right.

Another example: if we have a transformation $T : X \rightarrow Y$ and A which is a subset of X , then $T(A)$ is the image of A under T , which is equivalent to the set of transformations for each vector in A :



$$T(A) = \{T(\vec{x}) \in Y \mid \vec{x} \in A\}$$

Example: Image of a triangle.



$$T : X \rightarrow Y, \text{ where } A \subseteq X.$$

2.6.2 Image of a subspace

The image of a subspace under a transformation is also a subspace. That is, if V is a subspace, $T(V)$ is also a subspace.

2.6.3 Image of a transformation

If, instead of a subset or subspace, you take the transformation of an entire space, i.e. $T(\mathbb{R}^n)$, the terminology is different: that is called the *image of T* , notated $\text{im}(T)$.

Because we know matrix-vector products are linear transformations:

$$T(\vec{x}) = \mathbf{A}\vec{x}$$

The image of a linear transformation matrix \mathbf{A} is equivalent to its column space, that is:

$$\text{im}(T) = C(\mathbf{A})$$

2.6.4 Preimage of a set

The **preimage** is the inverse image. For instance, consider a transformation mapping from the domain X to the codomain Y :

$$T : X \rightarrow Y$$

And say you have a set S which is a subset of Y . You want to find the set of values in X which map to S , that is, the subset of X for which S is the image.

For a set S , this is notated:

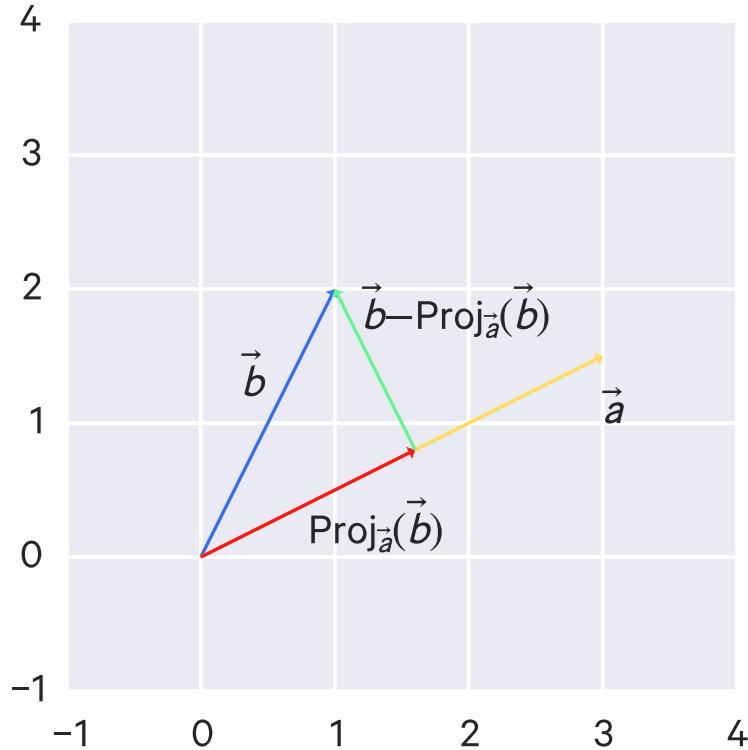
$$T^{-1}(S)$$

Note that not every point in S needs to map back to X . That is, S may contain some points for which there are no corresponding points in X . Because of this, the image of the preimage of S is not necessarily equivalent to S , but we can be sure that it is at least a subset:

$$T(T^{-1}(S)) \subseteq S$$

2.7 Projections

A **projection** can kind of be thought of as a “shadow” of a vector:



Here, we have the projection of \vec{x} - the red vector - onto the green line L . The projection is the dark red vector. This example is in \mathbb{R}^2 but this works in any \mathbb{R}^n .

The projection of \vec{x} onto line L is notated:

$$\text{Proj}_L(\vec{x})$$

More formally, a projection of a vector \vec{x} onto a line L is some vector in L where $\vec{x} - \text{Proj}_L(\vec{x})$ is orthogonal to L .

A line can be expressed as the set of all scalar multiples of a vector, i.e:

$$L = \{c\vec{v} \mid c \in \mathbb{R}\}$$

So we know that “some vector in L ” can be represented as $c\vec{v}$:

$$\text{Proj}_L(\vec{x}) = c\vec{v}$$

By our definition of a projection, we also know that $\vec{x} - \text{Proj}_L(\vec{x})$ is orthogonal to L , which can now be rewritten as:

$$(\vec{x} - c\vec{v}) \cdot \vec{v} = \vec{0}$$

(This is the definition of orthogonal vectors.)

Written in terms of c , this simplifies down to:

$$c = \frac{\vec{x} \cdot \vec{v}}{\vec{v} \cdot \vec{v}}$$

So then we can rewrite:

$$\text{Proj}_L(\vec{x}) = \frac{\vec{x} \cdot \vec{v}}{\vec{v} \cdot \vec{v}} \vec{v}$$

or, better:

$$\text{Proj}_L(\vec{x}) = \frac{\vec{x} \cdot \vec{v}}{||\vec{v}||^2} \vec{v}$$

And you can pick whatever vector for \vec{v} so long as it is part of line L .

However, if \vec{v} is a unit vector, then the projection is simplified even further:

$$\text{Proj}_L(\vec{x}) = (\vec{x} \cdot \hat{u}) \hat{u}$$

Projections are linear transformations (they satisfy the requirements, proof omitted), so you can represent them as matrix-vector products:

$$\text{Proj}_L(\vec{x}) = \mathbf{A}\vec{x}$$

where the transformation matrix \mathbf{A} is:

$$\mathbf{A} = \begin{bmatrix} u_1^2 & u_2 u_1 \\ u_1 u_2 & u_2^2 \end{bmatrix}$$

where u_i are components of the unit vector.

2.7.1 Projections onto subspaces

Given that V is a subspace of \mathbb{R}^n , we know that V^\perp is also a subspace of \mathbb{R}^n , and we have a vector \vec{x} such that $\vec{x} \in \mathbb{R}^n$, we know that $\vec{x} = \vec{v} + \vec{w}$ where $\vec{v} \in V$ and $\vec{w} \in V^\perp$, then:

$$\begin{aligned}\text{Proj}_V \vec{x} &= \vec{v} \\ \text{Proj}_{V^\perp} \vec{x} &= \vec{w}\end{aligned}$$

Reminder: a projection onto a subspace is the same as a projection onto a line (a line is a subspace):

$$\text{Proj}_V \vec{x} = \frac{\vec{x} \cdot \vec{v}}{\vec{v} \cdot \vec{v}} \vec{v}$$

where:

$$\begin{aligned}V &= \text{span}(\vec{v}) \\ V &= \{c\vec{v} \mid c \in \mathbb{R}\}\end{aligned}$$

So $\text{Proj}_V \vec{x}$ is the unique vector $\vec{v} \in V$ such that $\vec{x} = \vec{v} + \vec{w}$ where \vec{w} is a unique member of V^\perp .

Projection onto a subspace as a linear transform

$$\text{Proj}_V(\vec{x}) = \mathbf{A}(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \vec{x}$$

where V is a subspace of \mathbb{R}^n .

Note that $\mathbf{A}(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T$ is just some matrix, which we can call \mathbf{B} , so this is in the form of a linear transform, $\mathbf{B}\vec{x}$.

Also $\vec{v} = \text{Proj}_V \vec{x}$, so:

$$\vec{x} = \text{Proj}_V \vec{x} + \vec{w}$$

where \vec{w} is a unique member of V^\perp .

Projections onto subspaces with orthonormal bases

Given that V is a subspace of \mathbb{R}^n and $B = \{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k\}$ is an orthonormal basis for V . We have a vector $\vec{x} \in \mathbb{R}^n$, so $\vec{x} = \vec{v} + \vec{w}$ where $\vec{v} \in V$ and $\vec{w} \in V^\perp$.

We know (see previously) that by definition:

$$\text{Proj}_V(\vec{x}) = \mathbf{A}(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \vec{x}$$

which is quite complicated. It is much simpler for orthonormal bases:

$$\text{Proj}_V(\vec{x}) = \mathbf{A}\mathbf{A}^T\vec{x}$$

2.8 Identifying transformation properties

2.8.1 Determining if a transformation is surjective

A transformation $T(\vec{x}) = \mathbf{A}\vec{x}$ is surjective ("onto") if the column space of \mathbf{A} equals the codomain:

$$\text{span}(a_1, a_2, \dots, a_n) = C(\mathbf{A}) = \mathbb{R}^m$$

which can also be stated as:

$$\text{rank}(\mathbf{A}) = m$$

2.8.2 Determining if a transformation is injective

A transformation $T(\vec{x}) = \mathbf{A}\vec{x}$ is injective ("one-to-one") if the nullspace of \mathbf{A} contains only the zero vector:

$$N(\mathbf{A}) = \{\vec{0}\}$$

which is true if the set of \mathbf{A} 's column vectors is linearly independent. This can also be stated as:

$$\text{rank}(\mathbf{A}) = n$$

2.8.3 Determining if a transformation is invertible

A transformation is invertible if it is both injective and surjective.

For a transformation to be surjective:

$$\text{rank}(\mathbf{A}) = m$$

And for a transformation to be surjective:

$$\text{rank}(\mathbf{A}) = n$$

Therefore for a transformation to be invertible:

$$\text{rank}(\mathbf{A}) = m = n$$

So the transformation matrix \mathbf{A} must be a square matrix.

2.8.4 Inverse transformations of linear transformations

Inverse transformations are linear transformations if the original transformation is both linear and invertible. That is, if T is invertible and linear, T^{-1} is linear:

$$\begin{aligned} T^{-1}(\vec{x}) &= \mathbf{A}^{-1}\vec{x} \\ (T^{-1} \circ T)(\vec{x}) &= \mathbf{A}^{-1}\mathbf{A}\vec{x} = I_n\vec{x} = \mathbf{A}\mathbf{A}^{-1}\vec{x} = (T \circ T^{-1})(\vec{x}) \end{aligned}$$

2.9 Eigenvalues and Eigenvectors

Say we have a linear transformation $T : \mathbb{R}^n \rightarrow \mathbb{R}^n$:

$$T(\vec{v}) = \mathbf{A}\vec{v} = \lambda\vec{v}$$

That is, \vec{v} is scaled by a transformation matrix λ .

We say that:

- \vec{v} is the **eigenvector** for T
- λ is the **eigenvalue** associated with that eigenvector

Eigenvectors are vectors for which matrix multiplication is equivalent to only a scalar multiplication, nothing more. λ , the **eigenvalue**, is the scalar that the transformation matrix \mathbf{A} is equivalent to.

Another way to put this: given a square matrix $A \in \mathbb{R}^{n \times n}$, we say $\lambda \in \mathbb{C}$ is an **eigenvalue** of A and $x \in \mathbb{C}^n$ is the corresponding **eigenvector** if:

$$Ax = \lambda x, x \neq 0$$

Note that \mathbb{C} refers to the set of complex numbers.

So this means that multiplying A by x just results in a new vector which points in the same direction has x but scaled by a factor λ .

For any eigenvector $x \in \mathbb{C}^n$ and a scalar $t \in \mathbb{C}$, $A(cx) = cAx = c\lambda x = \lambda(cx)$, that is, cx is also an eigenvector - but when talking about “the” eigenvector associated with λ , it is assumed that the eigenvector is normalized to length 1 (though you still have the ambiguity that both x and $-x$ are eigenvectors in this sense).

Eigenvalues and eigenvectors come up when maximizing some function of a matrix.

So what are our eigenvectors? What \vec{v} satisfies:

$$\mathbf{A}\vec{v} = \lambda\vec{v}, \vec{v} \neq 0$$

We can do:

$$\begin{aligned}\mathbf{A}\vec{v} &= \lambda\vec{v} \\ \vec{0} &= \lambda\vec{v} - \mathbf{A}\vec{v}\end{aligned}$$

We know that $\vec{v} = \mathbf{I}_n\vec{v}$, so we can do:

$$\vec{0} = \lambda\mathbf{I}_n\vec{v} - \mathbf{A}\vec{v} = (\lambda\mathbf{I}_n - \mathbf{A})\vec{v}$$

The first term, $\lambda\mathbf{I}_n - \mathbf{A}$, is just some matrix which we can call \mathbf{B} , so we have:

$$\vec{0} = \mathbf{B}\vec{v}$$

which, by our definition of nullspace, indicates that \vec{v} is in the nullspace of \mathbf{B} . That is:

$$\vec{v} \in N(\lambda\mathbf{I}_n - \mathbf{A})$$

2.9.1 Properties of eigenvalues and eigenvectors

- The trace of A is equal to the sum of its eigenvalues: $\text{tr}(A) = \sum_{i=1}^n \lambda_i$.
- The determinant of A is equal to the product of its eigenvalues: $|A| = \prod_{i=1}^n \lambda_i$.
- The rank of A is equal to the number of non-zero eigenvalues of A .
- If A is non-singular, then $\frac{1}{\lambda_i}$ is an eigenvalue of A^{-1} with associated eigenvector x_i , i.e. $A^{-1}x_i = \left(\frac{1}{\lambda_i}\right)x_i$.

- The eigenvalues of a diagonal matrix $D = \text{diag}(d_1, \dots, d_n)$ are just the diagonal entries d_1, \dots, d_n .

2.9.2 Diagonalizable matrices

All eigenvector equations can be written simultaneously as:

$$AX = X\Lambda$$

where the columns of $X \in \mathbb{R}^{n \times n}$ are the eigenvectors of A and Λ is a diagonal matrix whose entries are the eigenvalues of A , i.e. $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$.

If the eigenvectors of A are linearly independent, then the matrix X will be invertible, so that $A = X\Lambda X^{-1}$. A matrix that can be written in this form is called **diagonalizable**.

2.9.3 Eigenvalues & eigenvectors of symmetric matrices

For a symmetric matrix $A \in \mathbb{S}^n$, the eigenvalues of A are all real and the eigenvectors of A are all orthonormal.

If for all of A 's eigenvalues λ_i ...

- $\lambda_i > 0$, then A is positive definite.
- $\lambda_i \geq 0$, then A is positive semidefinite.
- $\lambda_i < 0$, then A is negative definite.
- $\lambda_i \leq 0$, then A is negative semidefinite.
- have both positive and negative values, then A is indefinite.

Example

Say we have a linear transformation $T(\vec{x}) = \mathbf{A}\vec{x}$. Here are some example values of \vec{x} being input and the output vectors they yield (it's not important here what \mathbf{A} actually looks like, its just to help distinguish what is and isn't an eigenvector.)

- $\mathbf{A} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
 - \vec{x} is not an eigenvector, it was not merely scaled by \mathbf{A} .
- $\mathbf{A} \begin{bmatrix} 4 \\ 7 \end{bmatrix} = \begin{bmatrix} 8 \\ 14 \end{bmatrix}$
 - \vec{x} is an eigenvector, it was only scaled by \mathbf{A} . This is a simple example where the vector was scaled up by 2, so the eigenvalue here is 2.

2.9.4 Eigenspace

The eigenvectors that correspond to an eigenvalue λ form the **eigenspace** for that λ , notated E_λ :

$$E_\lambda = N(\lambda \mathbf{I}_n - \mathbf{A})$$

2.9.5 Eigenbasis

Say we have an $n \times n$ matrix \mathbf{A} . An **eigenbasis** is a basis for \mathbb{R}^n consisting entirely of eigenvectors for \mathbf{A} .

3

Calculus

3.1 Differentiation

The *slope* of a function (more generally called the **gradient**) can be thought of as the *rate of change* for that function.

For a linear function $f(x) = ax + b$, a is the slope; it is constant for all x throughout the function.

But for non-linear functions, e.g. $f(x) = 3x^2$, the slope varies along with x .

Differentiation is a way to find another function, called the **derivative** of the original function, that gives us the rate of change (slope) of one variable with respect to another variable.

Example

We have a car and have a variable x which describes its position at a given point in time t . That is, $f(t) = x$.

With differentiation we can get $\frac{dx}{dt}$ which is the rate of change of the car's position wrt to time, i.e. the speed (velocity) of the car.

Note that this is *not* the same as $\frac{\Delta x}{\Delta t}$, which gives us the change in x over a time interval Δt . This is the average velocity over that time interval.

If instead we want instantaneous velocity - the velocity at a given point in time - we need to have the time interval Δt approach 0 (we can't set Δt to 0 because then we have division by 0). This is equivalent to the derivative described previously:

$$\lim_{\Delta t \rightarrow 0} \frac{\Delta x}{\Delta t} = \frac{dx}{dt}$$

This can be read as

- “the rate of change in x with respect to t ”, or
- “an infinitesimal value of y divided by an infinitesimal value of x ”

For a given function $f(x)$, this can also be written as:

$$\frac{d}{dx} f(x) = \lim_{\Delta \rightarrow 0} \frac{f(x + \Delta) - f(x)}{\Delta}$$

3.1.1 Formal definition of a derivative

Let $f(x)$ be a function. Then:

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

if this limit exists. In which case, f is **differentiable** at x and its **derivative** at x is $f'(x)$.

Basically what this is saying is: take two points along your line and calculate the slope between them. As $\Delta x \rightarrow 0$, these points get closer and closer until they are basically the same point, and you calculate the slope there - this is approximately the slope at that single point.

3.1.2 Notation

A derivative of a function $y = f(x)$ may be notated:

- $f'(x)$
- $D_x[f(x)]$
- $Df(x)$
- $\frac{dy}{dx}$
- $\frac{d}{dx}[y]$

As a special case, if we are looking at a variable with respect to time t , we can use Newton's dot notation:

- $\dot{f} = \frac{df}{dt}$

3.1.3 Differentiation rules

- **Derivative of a constant function:** For any fixed real number c , $\frac{d}{dx}[c] = 0$.
 - This is because a constant function is just a horizontal line (it has a slope of 0).
- **Derivative of a linear function:** For any fixed real numbers m and c , $\frac{d}{dx}[mx + c] = m$

- **Constant multiple rule:** For any fixed real number c , $\frac{d}{dx}[cf(x)] = c\frac{d}{dx}[f(x)]$
- **Addition rule:** $\frac{d}{dx}[f(x) \pm g(x)] = \frac{d}{dx}[f(x)] \pm \frac{d}{dx}[g(x)]$
- **The power rule:** $\frac{d}{dx}[x^n] = nx^{n-1}$
- **Product rule:** $\frac{d}{dx}[f(x) \cdot g(x)] = f(x) \cdot g'(x) + f'(x) \cdot g(x)$
- **Quotient rule:** $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{g(x)^2}$

Example

$$\frac{d}{dx}[6x^5 + 3x^2 + 3x + 1]$$

- Apply the addition rule: $\frac{d}{dx}[6x^5] + \frac{d}{dx}[3x^2] + \frac{d}{dx}[3x] + \frac{d}{dx}[1]$
- Apply the linear and constant rules: $\frac{d}{dx}[6x^5] + \frac{d}{dx}[3x^2] + 3 + 0$
- Apply the constant multiplier rule: $6\frac{d}{dx}[x^5] + 3\frac{d}{dx}[x^2] + 3$
- Then the power rule: $6(5x^4) + 3(2x) + 3$
- And finally: $30x^4 + 6x + 3$

Chain rule

If a function f is composed of two differentiable functions $y(x)$ and $u(x)$, so that $f(x) = y(u(x))$, then $f(x)$ is differentiable and:

$$\frac{df}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

This rule can be applied sequentially to nestings (compositions) of many functions:

$$f(g(h(x))) \frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dx}$$

The chain rule is very useful when you recompose functions in terms of nested functions.

Example

Given $f(x) = (x^2 + 1)^3$, we can define another function $u(x) = x^2 + 1$, thus we can rewrite $f(x)$ in terms of $u(x)$, that is: $f(x) = u(x)^3$.

- We can apply the chain rule: $\frac{df}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$.
- Then substitute: $\frac{df}{dx} = \frac{d}{du}[u^3] \cdot \frac{d}{dx}(x^2 + 1)$.
- Then we can just apply the rest of our rules: $\frac{df}{dx} = 3u^2 \cdot 2x$.
- Then substitute again: $\frac{df}{dx} = 3(x^2 + 1)^2 \cdot 2x$ and simplify.

3.1.4 Higher order derivatives

The derivative of a function as described above is the **first derivative**.

The **second derivative**, or **second order derivative**, is the derivative of the first derivative, denoted $f''(x)$.

There's also the third derivative, $f'''(x)$, and a fourth, and so on.

Any derivative beyond the first is a **higher order derivative**.

Notation

The above notation gets unwieldy, so there are alternate notations.

For the n^{th} derivative:

- $f^{(n)}(x)$ (this is to distinguish from $f^n(x)$ which is the quantity $f(x)$ raised to the n^{th} power)
- $\frac{d^n f}{dx^n}$ (Leibniz notation)
- $\frac{d^n}{dx^n}[f(x)]$ (another form of Leibniz notation)
- $D^n f$ (Euler's notation)

3.1.5 Explicit differentiation

When dealing with multiple variables, there is sometimes the option of **explicit differentiation**. This simply involves expressing one variable in terms of the other.

For example: $x^2 + y^2 = 1$. This can be rewritten in terms of x like so: $y = \pm(1 - x^2)^{1/2}$.

Here it is easy to apply the chain rule:

$$\begin{aligned}
 u(x) &= 1 - x^2 \\
 y &= u(x)^{1/2} \\
 \frac{dy}{dx} &= \frac{d}{du}[u^{1/2}] \cdot \frac{d}{dx}[1 - x^2] \\
 &= \frac{d}{du}[u^{1/2}] \cdot \left(\frac{d}{dx}[1] - \frac{d}{dx}[x^2]\right) \\
 &= \frac{d}{du}[u^{1/2}] \cdot \left(-\frac{d}{dx}[x^2]\right) \\
 &= \frac{d}{du}[u^{1/2}] \cdot (-2x) \\
 &= \frac{1}{2}u^{-1/2} \cdot (-2x) \\
 &= \frac{1}{2}(1 - x^2)^{-1/2} \cdot (-2x) \\
 &= -x(1 - x^2)^{-1/2} \\
 &= -\frac{x}{(1 - x^2)^{1/2}} \\
 &= -\frac{x}{y}
 \end{aligned}$$

3.1.6 Implicit differentiation

Implicit differentiation is useful for differentiating equations which cannot be explicitly differentiated because it is impossible to isolate variables. With implicit differentiation, you do not need to define one of the variables in terms of the other.

For example, using the same equation from before: $x^2 + y^2 = 1$.

First, differentiate with respect to x on both sides of the equation:

$$\begin{aligned}\frac{d}{dx}[x^2 + y^2] &= \frac{d}{dx}[1] \\ \frac{d}{dx}[x^2 + y^2] &= 0 \\ \frac{d}{dx}[x^2] + \frac{d}{dx}[y^2] &= 0\end{aligned}$$

To differentiate $\frac{d}{dx}[y^2]$, we can define a new function $f(y(x)) = y^2$ and then apply the chain rule:

$$\frac{df}{dx} = \frac{df}{dy} \cdot \frac{dy}{dx} = \frac{d}{dy}[y^2] \cdot \frac{dy}{dx} = 2y \cdot y'$$

So returning to our other in-progress derivative:

$$\frac{d}{dx}[x^2] + \frac{d}{dx}[y^2] = 0$$

We can substitute and bring it to completion:

$$\begin{aligned}\frac{d}{dx}[x^2] + \frac{d}{dx}[y^2] &= 0 \\ \frac{d}{dx}[x^2] + 2yy' &= 0 \\ 2x + 2yy' &= 0 \\ 2yy' &= -2x \\ y' &= -\frac{2x}{2y} \\ y' &= -\frac{x}{y}\end{aligned}$$

3.1.7 Derivatives of trigonometric functions

$$\begin{aligned}
 \frac{d}{dx} \sin(x) &= \cos(x) \\
 \frac{d}{dx} \cos(x) &= -\sin(x) \\
 \frac{d}{dx} \tan(x) &= \sec^2(x) \\
 \frac{d}{dx} \sec(x) &= \sec(x) \tan(x) \\
 \frac{d}{dx} \csc(x) &= -\csc(x) \cot(x) \\
 \frac{d}{dx} \cot(x) &= -\csc^2(x) \\
 \frac{d}{dx} \arcsin(x) &= \frac{1}{\sqrt{1-x^2}} \\
 \frac{d}{dx} \arccos(x) &= \frac{-1}{\sqrt{1-x^2}} \\
 \frac{d}{dx} \arctan(x) &= \frac{1}{1+x^2}
 \end{aligned}$$

3.1.8 Derivatives of exponential and logarithmic functions

$$\begin{aligned}
 \frac{d}{dx} e^x &= e^x \\
 \frac{d}{dx} a^x &= \ln(a) a^x \\
 \frac{d}{dx} \ln(x) &= \frac{1}{x} \\
 \frac{d}{dx} \log_b(x) &= \frac{1}{x \ln(b)}
 \end{aligned}$$

3.1.9 Extreme Value Theorem

A **global maximum** (or *absolute maximum*) of a function f on a closed interval I is a value $f(c)$ such that $f(c) \geq f(x)$ for all x in I .

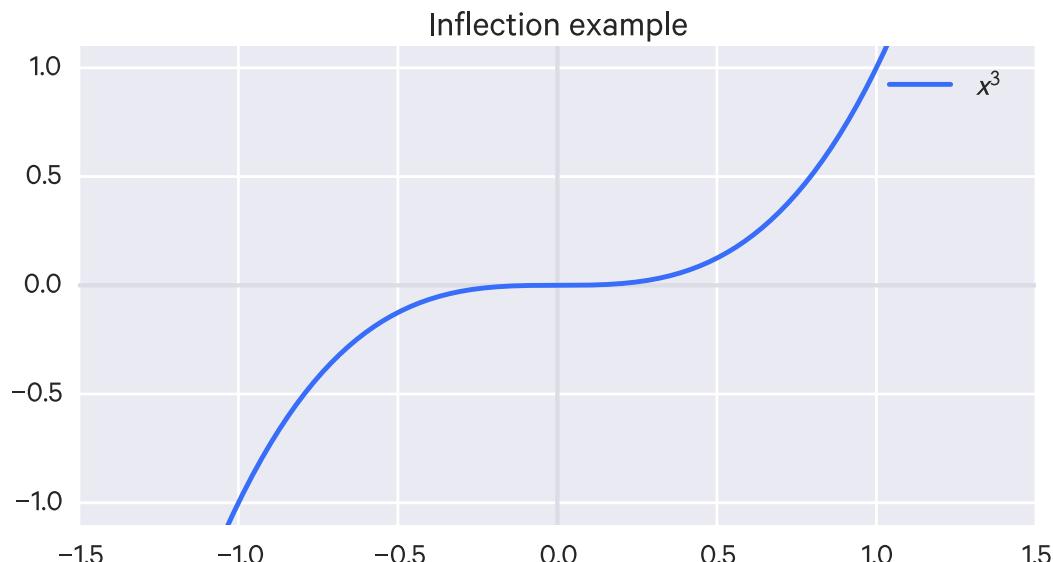
A **global minimum** (or *absolute minimum*) of a function f on a closed interval I is a value $f(c)$ such that $f(c) \leq f(x)$ for all x in I .

The **extreme value theorem** states that if f is a function that is continuous on the closed interval $[a, b]$, then f has both a global minimum and a global maximum on $[a, b]$. It is assumed that a and b are both finite.

Extrema and inflection/inflexion points

Note that at *any extremum* (i.e. a minimum or a maximum), global or local, the slope is 0 because the graph stops rising/falling and “turns around”. For this reason, extrema are also called **stationary points** or *turning points*.

Thus, the first derivative of a function is equal to 0 at extrema. But the converse does not hold true: the first derivative of a function is not always an extrema when it equals 0. This is because a slope of 0 may also be found at a point of **inflection**:



An example of inflection.

To discern extrema from inflection points, you can use the *extremum test*, aka the *2nd derivative test*.

If the 2nd derivative at the stationary point is positive (increasing) or negative (decreasing), then we know we have a minimum or a maximum, respectively.

However, if the 2nd derivative is also 0, then we still have not distinguished the point. What you can do is continue differentiating until you get a non-zero result.

If we take n to be the order of the derivative yielding the non-zero result, then if $n - 1$ is odd, we have a true extremum. Again, if the non-zero result is positive, then it is a minimum, if it is negative, it is maximum.

However, if $n - 1$ is even, then we have a point of inflection.

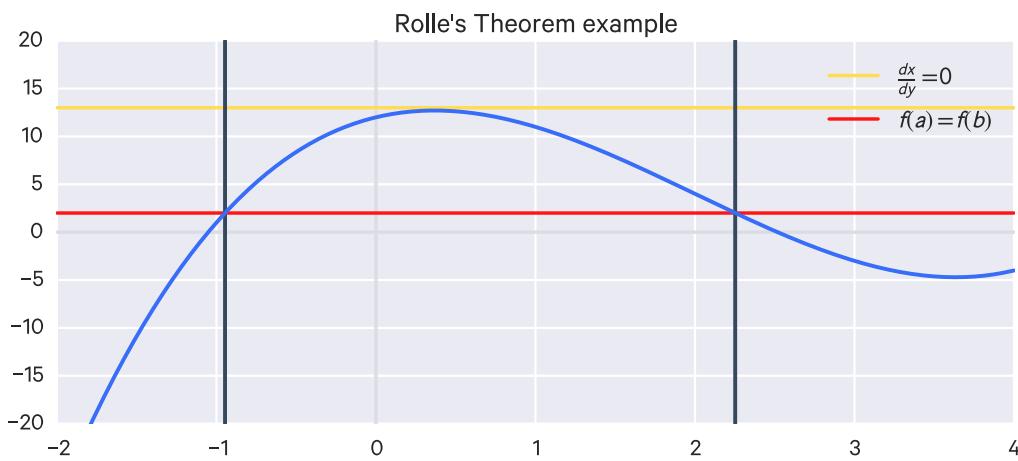
Critical points

A **critical point** are points where the function's derivative are 0 or not defined. So stationary points are critical points.

3.1.10 Rolle's Theorem

If a function $f(x)$ is continuous on the closed interval $[a, b]$, is differentiable on the open interval (a, b) , and $f(a) = f(b)$, then there exists at least one number c in the interval (a, b) such that $f'(c) = 0$.

This is basically saying that if you have an interval which ends with the same value it starts with, at some point in that curve the slope will be 0:



3.1.11 Mean Value Theorem

If $f(x)$ is continuous on the closed interval $[a, b]$ and differentiable on the open interval (a, b) , there exists a number c in the open interval (a, b) such that

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

This is basically saying that there is some point on the interval where its instantaneous slope is equal to the average slope of the interval.

Rolle's Theorem is a special case of the Mean Value Theorem where $f(a) = f(b)$.

3.1.12 L'Hopital's Rule

An **indeterminate limit** is one which results in $\frac{0}{0}$ or $\frac{\pm\infty}{\pm\infty}$.

If $\lim_{x \rightarrow c} \frac{f(x)}{g(x)}$ is indeterminate of type $\frac{0}{0}$ or $\frac{\pm\infty}{\pm\infty}$, then $\lim_{x \rightarrow c} \frac{f(x)}{g(x)} = \lim_{x \rightarrow c} \frac{f'(x)}{g'(x)}$.

If the resulting limit here is also indeterminate, you can re-apply L'Hopital's rule until it is not.

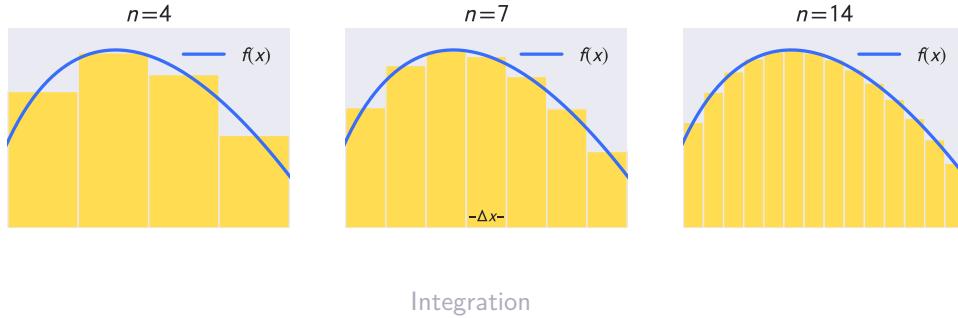
Note that c can be a finite value, ∞ , or $-\infty$.

3.2 Integration

3.2.1 Definite integral

How can we find the area under a graph?

We can try to approximate the area using a finite number (n) of rectangles. The area of rectangles are easy to calculate so we can just add up their area.



Integration

The more rectangles (i.e. increasing n) we fit, the better the approximation.

So we can have $n \rightarrow \infty$ to get the best approximation of the area under the curve.

Say we have a function $f(x)$ that is positive over some interval $[a, b]$. The width of each rectangle over that interval, divided into n rectangles (subintervals), is $\Delta x = \frac{b-a}{n}$. The endpoint of an subinterval can be denoted x_i , for $i = 0, 1, \dots, n$. For each i^{th} subinterval, we pick some sample point x_i^* in the interval $[x_{i-1}, x_i]$. This sample point is the height of the i^{th} rectangle.

Thus, for the i^{th} rectangle, we have as its area:

$$a_i = x_i^* \frac{b-a}{n}$$

or

$$a_i = x_i^* \Delta x$$

So the total area for the interval is:

$$A_n = \sum_{i=1}^n f(x_i^*) \Delta x$$

This kind of area approximation is called a **Riemann sum**.

The best approximation then is:

$$\lim_{n \rightarrow \infty} \sum_{i=1}^n f(x_i^*) \Delta x$$

So we define the **definite integral**:

Suppose f is a continuous function on $[a, b]$ and $\Delta x = \frac{b-a}{n}$. Then the definite integral of f between a and b is:

$$\int_a^b f(x)dx = \lim_{n \rightarrow \infty} A_n = \lim_{n \rightarrow \infty} \sum_{i=1}^n f(x_i^*) \Delta x$$

where x_i^* are any sample points in the interval $[x_{i-1}, x_i]$ and $x_k = a + k \cdot \Delta x$ for $k = 0, \dots, n$.

In the expression $\int_a^b f(x)dx$, f is the **integrand**, a is the **lower limit** and b is the **upper limit** of integration.

Left-handed vs right-handed Riemann sums

A right-handed Riemann sum is just one where $x_i^* = x_i$, and a left-handed Riemann sum is just one where $x_i^* = x_{i-1}$.

3.2.2 Basic properties of the integral

- **The constant rule:** $\int_a^b cf(x)dx = c \int_a^b f(x)dx$
 - A special case rule for integrating constants is: $\int_a^b c dx = c(b - a)$
- **Addition and subtraction rule:** $\int_a^b (f(x) \pm g(x))dx = \int_a^b f(x)dx \pm \int_a^b g(x)dx$
- **The comparison rule**
 - Suppose $f(x) \geq 0$ for all x in $[a, b]$. Then $\int_a^b f(x)dx \geq 0$.
 - Suppose $f(x) \geq g(x)$ for all x in $[a, b]$. Then $\int_a^b f(x)dx \geq \int_a^b g(x)dx$.
 - Suppose $M \geq f(x) \geq m$ for all x in $[a, b]$. Then $M(b - a) \geq \int_a^b f(x)dx \geq m(b - a)$.
- **Additivity with respect to endpoints:** Suppose $a < c < b$. Then $\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$.
 - This is basically saying the area under the graph from a to b is equal to the area under the graph from a to c plus the area under the graph from c to b , so long as c is some point between a and b .
- **Power rule of integration:** As long as $n \neq 1$ and $0 \notin [a, b]$, or $n > 0$, $\int_a^b x^n dx = \frac{x^{n+1}}{n+1} \Big|_a^b = \frac{b^{n+1} - a^{n+1}}{n+1}$

3.2.3 Mean Value Theorem for Integration

Suppose $f(x)$ is continuous on $[a, b]$. Then $\frac{\int_a^b f(x)dx}{b-a} = f(c)$ for some c in $[a, b]$.

3.2.4 The fundamental theorem of calculus

The fundamental theorem of calculus connects the concept of a derivative to that of an integral.

Suppose that f is continuous on $[a, b]$. We can define a function F like so:

$$F(x) = \int_a^x f(t)dt \text{ for } x \in [a, b]$$

Part I

Suppose f is continuous on $[a, b]$ and F is defined by $F(x) = \int_a^x f(t)dt$.

Then F is differentiable on (a, b) for all $x \in (a, b)$, i.e.:

$$F'(x) = f(x)$$

When we have some functions F and f where $F'(x) = f(x)$ for every x in some interval I , then F is the **antiderivative** of f on I .

Part II

Suppose f is continuous on $[a, b]$ and F is any antiderivative of f . Then:

$$\int_a^b f(x)dx = F(b) - F(a)$$

Note that $F(b) - F(a)$ may be notated as $F(x)|_a^b$.

3.2.5 Indefinite integral

F is an antiderivative of f if $F'(x) = f(x)$. But F is not the only antiderivative: any constant can be added to F without changing the derivative.

Thus we define the **indefinite integral** as:

$$\int f(x)dx = F(x) + C$$

where F satisfies $F'(x) = f(x)$ and C is any constant.

Basic properties of indefinite integrals

The integral rules defined above still apply.

- **Power rule for indefinite integrals:** For all $n \neq -1$, $\int x^n dx = \frac{1}{n+1}x^{n+1} + C$
- **Integral of the inverse function:** For $f(x) = \frac{1}{x}$, remember that $\frac{d}{dx} \ln x = \frac{1}{x}$, so $\int \frac{dx}{x} = \ln|x| + C$
- **Integral of the exponential function:** Because $\frac{d}{dx} e^x = e^x$, $\int e^x dx = e^x + C$
- **The substitution rule for indefinite integrals:** Assume u is differentiable with a continuous derivative and that f is continuous on the range of u . Then $\int f(u(x)) \frac{du}{dx} dx = \int f(u) du$.
 - Remember that $\frac{du}{dx}$ is *not* a fraction, so you're not just “canceling” things out here.

Integration by parts

Suppose f and g are differentiable and their derivatives are continuous. Then:

$$\int f(x)g(x)dx = \left(f(x) \int g(x)dx \right) - \int \left(f'(x) \int g(x)dx \right) dx$$

You set $f(x)$ in the following order, called *ILATE*:

- I for inverse trigonometric functions
- L for log functions
- A for algebraic functions
- T for trigonometric functions
- E for exponential functions

3.2.6 Improper integrals

Say we have the integral

$$\int_1^\infty \frac{dx}{x^2}$$

If we set the upper bound to be a finite value b and have it approach infinity, we get:

$$\begin{aligned} \lim_{b \rightarrow \infty} \int_1^b \frac{dx}{x^2} &= \lim_{b \rightarrow \infty} \left(\frac{1}{1} - \frac{1}{b} \right) \\ &= \lim_{b \rightarrow \infty} \left(1 - \frac{1}{b} \right) \\ &= 1 \end{aligned}$$

The formal definition:

1. Suppose $\int_a^b f(x)dx$ exists for all $b \geq a$. Then we define

$$\int_a^\infty f(x)dx = \lim_{b \rightarrow \infty} \int_a^b f(x)dx$$

as long as this limit exists and is finite. If it does exist we say the integral is *convergent* and otherwise we say it is *divergent*.

2. Similarly if $\int_a^b f(x)dx$ exists for all $a \leq b$ we define

$$\int_{-\infty}^b f(x)dx = \lim_{a \rightarrow -\infty} \int_a^b f(x)dx$$

3. Finally, suppose c is a fixed real number and that $\int_{-\infty}^c f(x)dx$ and $\int_c^\infty f(x)dx$ are both convergent. Then we define

$$\int_{-\infty}^\infty f(x)dx = \int_{-\infty}^c f(x)dx + \int_c^\infty f(x)dx$$

Improper integrals with a finite number of discontinuities

Suppose f is continuous on $[a, b]$ except at points $c_1 < c_2 < \dots < c_n$ in $[a, b]$.

We define

$$\int_a^b f(x)dx = \int_a^{c_1} f(x)dx + \int_{c_1}^{c_2} f(x)dx + \dots + \int_{c_n}^b f(x)dx$$

as long as each integral on the right converges.

Improper integral with one discontinuity

As a simpler example, say we have an improper integral with a single discontinuity.

If f is continuous on the interval $[a, b)$ and is discontinuous at b , we define

$$\int_a^b f(x)dx = \lim_{c \rightarrow b^-} \int_a^c f(x)dx$$

If this limit exists, the integral we say it converges and otherwise we say it diverges.

Similarly, if f is continuous on the interval $(a, b]$ and is discontinuous at a , we define

$$\int_a^b f(x)dx = \lim_{c \rightarrow a^+} \int_a^c f(x)dx$$

Finally, if f has a discontinuity at a point c in (a, b) and is continuous at all other points in $[a, b]$, if both $\int_a^c f(x)dx$ and $\int_c^b f(x)dx$ converge, we define

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$$

3.3 Multivariable Calculus

We are frequently dealing with data in many dimensions, so we must expand the previous concepts of derivatives and integrals to higher-dimensional spaces.

3.3.1 Integration

Double integrals

A definite integral for $y = f(x)$ is the area under the curve of $f(x)$, which is the sum of the areas of infinitely small rectangles assembled in the shape of the curve.

But say we are working with three dimensions, i.e. we have $z = f(x, y)$. Then the *volume under the surface of $f(x, y)$* is the sum of the volumes of infinitely small chunks in the shape of the surface.

The area of one face of that chunk is the area under the curve, with respect to x , from $x = 0$ to $x = b$ (in the illustration below), i.e. the integral:

$$\int_0^b f(x, y)dx$$

Because this is with respect to x , this integral will be some function of y , e.g. $g(y)$.

To get the volume of this chunk, we multiply that area by some depth dy , so the volume of a chunk is:

$$\left(\int_0^b f(x, y)dx \right) dy$$

So if we want to get the volume in the bounds of $y = 0$, $y = a$, then we integrate again:

$$\int_0^a \left(\int_0^b f(x, y)dx \right) dy$$

A double integral!

It is also written without the parentheses:

$$\int_0^a \int_0^b f(x, y) dx dy$$

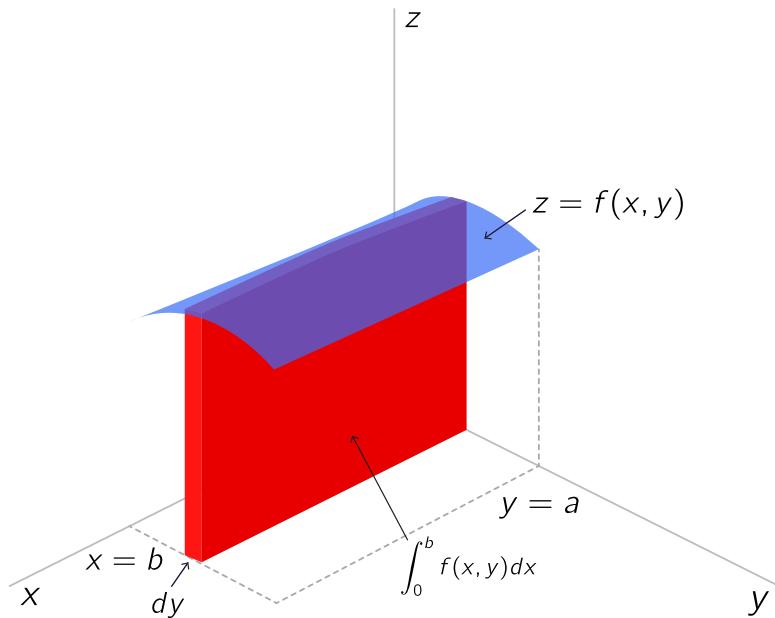


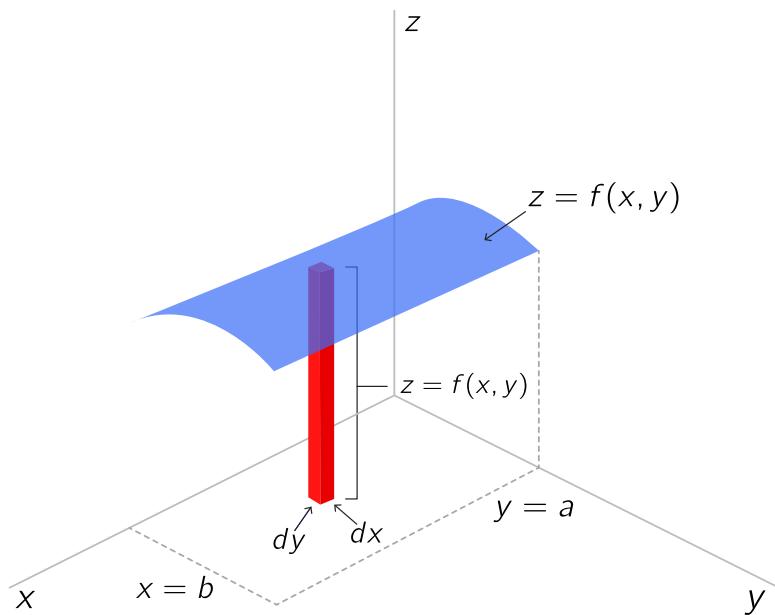
Illustration of a double integral.

Note that here we first integrated wrt to x and then y , but you can do it the other way around as well (integrate wrt y first, then x).

Note: the lower bounds here were 0 but that's just an example.

Another way of conceptualizing double integrals

You could instead conceptualize the double integral as the sum of the volumes of infinitely small columns:



Another illustration of a double integral.

The area of each column's base, $dx \cdot dy$, is sometimes notated as dA .

Variable boundaries

In the previous example we had fixed boundaries (see accompanying illustration, on the left).

What if instead we have a variable boundary (see accompanying illustration, on the right. The lower x boundary varies now).

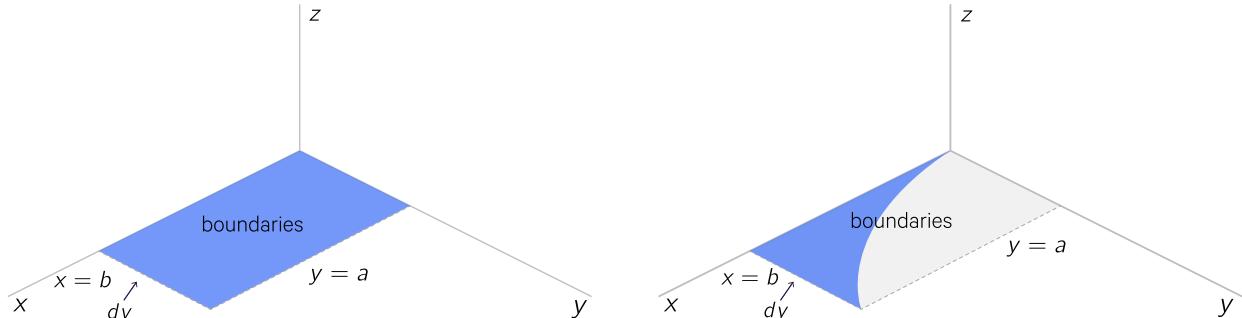


Illustration of variable boundaries.

Well you express variable boundaries as a functions. As is the case in the example above, the lower x boundary is some function of y , $g(y)$. So the volume would be:

$$\int_0^a \int_{g(y)}^b f(x, y) dx dy$$

That's if you first integrate wrt to x . If you first integrate wrt to y , instead the upper y boundary is varying and that would be some function of x , $h(x)$, i.e.:

$$\int_0^b \int_0^{h(x)} f(x, y) dy dx$$

Triple integrals

Triple integrals also involve infinitely small volumes and in many cases are no different than double integrals.

So why use triple integrals? Well they are good for calculating the *mass* of something - if the density under the surface is not uniform. The density at a given point is expressed as $f(x, y, z)$, so the mass of a variably dense volume can be expressed as:

$$\int_{x_0}^{X_{final}} \int_{y_0}^{Y_{final}} \int_{z_0}^{Z_{final}} f(x, y, z) dz dy dx$$

3.3.2 Partial derivatives

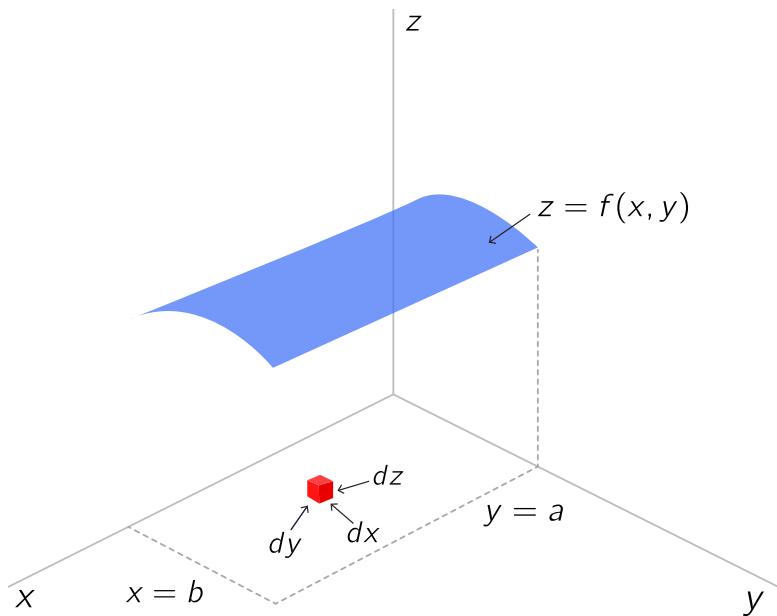


Illustration of a triple integral.

Say you have a function $z = f(x, y)$.

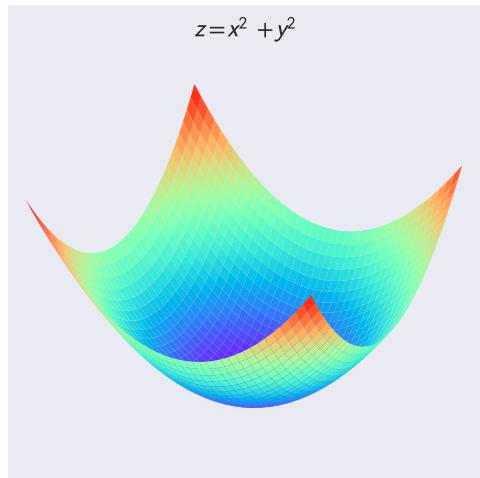
With two variables, we are now working in three dimensions. How does differential calculus work in 3 (or more) dimensions? In three dimensions, what is the slope at a given point? Any given point has an infinite number of tangent lines (only one tangent plane though). So when you take a derivative in three dimensions, you have to specify what direction that derivative is in.

Say we have $z = x^2 + xy + y^2$. If we want to take a derivative of this function, we have to hold one variable constant and derive with respect to the other variable. This derivative is called a **partial derivative**. If we were doing it wrt to x , then it would be notated as:

$$\frac{\partial z}{\partial x} \quad \text{or} \quad f_x(x, y)$$

So we could work this out as:

$$\begin{aligned} y &= C \\ z &= x^2 + xC + C^2 \\ \frac{\partial z}{\partial x} &= 2x + C \\ \frac{\partial z}{\partial x} &= 2x + y \end{aligned}$$

A 3d function (the sphere function, $z = x^2 + y^2$)

Then you could get the partial derivative wrt to y , i.e.:

$$\begin{aligned}x &= C \\z &= C^2 + Cy + y^2 \\\frac{\partial z}{\partial y} &= C + 2y \\\frac{\partial z}{\partial y} &= x + 2y\end{aligned}$$

The plane that these two functions define together for a given point (x, y) is the tangent plane at that point.

More generally, for a function $f(x, y)$, the partial derivatives would be:

$$\begin{aligned}\frac{\partial}{\partial x} f(x, y) &= \lim_{\Delta \rightarrow 0} \frac{f(x + \Delta, y) - f(x, y)}{\Delta} \\\frac{\partial}{\partial y} f(x, y) &= \lim_{\Delta \rightarrow 0} \frac{f(x, y + \Delta) - f(x, y)}{\Delta}\end{aligned}$$

3.3.3 Directional derivatives

Partial derivatives can be generalized into *directional derivatives*, which are derivatives with respect to any arbitrary line (it does not have to be, for example, with respect to the x or y axis). That is, with respect to any arbitrary direction.

3.3.4 Gradients

A gradient is a vector of all the partial derivatives at a given point.

The gradient of a function f is notated:

$$\nabla f$$

Sometimes it is just notated as ∇ .

The gradient of some function $f(x, y)$, i.e. $z = f(x, y)$ is:

$$\nabla f = f_x \hat{i} + f_y \hat{j}$$

That is, the partial derivative of f wrt to x times the unit vector in the x direction, \hat{i} , plus the partial of f wrt to y times the unit vector in the y direction, \hat{j} .

It can also be written (this is just different notation):

$$\nabla f = \frac{\partial}{\partial x} f(x, y) \hat{i} + \frac{\partial}{\partial y} f(x, y) \hat{j}$$

It's worth noting that this can be thought of in terms of matrices, i.e. given some function $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$ (that is, it takes a matrix $A \in \mathbb{R}^{m \times n}$ and returns a real value), then the gradient of f , with respect to the matrix A , is the matrix of partial derivatives:

$$\nabla_A f(A) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{11}} & \frac{\partial f(A)}{\partial A_{12}} & \cdots & \frac{\partial f(A)}{\partial A_{1n}} \\ \frac{\partial f(A)}{\partial A_{21}} & \frac{\partial f(A)}{\partial A_{22}} & \cdots & \frac{\partial f(A)}{\partial A_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(A)}{\partial A_{m1}} & \frac{\partial f(A)}{\partial A_{m2}} & \cdots & \frac{\partial f(A)}{\partial A_{mn}} \end{bmatrix}$$

Which is to say that $(\nabla_A f(A))_{ij} = \frac{\partial f(A)}{\partial A_{ij}}$.

Properties

Some properties, taken from equivalent properties of partial derivatives, are:

- $\nabla_x(f(x) + g(x)) = \nabla_x f(x) + \nabla_x g(x)$
- For $t \in \mathbb{R}$, $\nabla_x(tf(x)) = t\nabla_x f(x)$

Example

Say we have the function $f(x, y) = x^2 + xy + y^2$.

Using the partials we calculated previously, the gradient is:

$$\nabla f = (2x + y)\hat{i} + (2y + x)\hat{j}$$

So what we're really calculating here is a *vector field*, which gives an x and a y vector, with the magnitude of the partial derivative of f wrt to x and the partial derivative of f wrt to y , respectively, then getting the vector which is the sum of those two vectors.

What the gradient tells us is, for a given point, what direction to travel to get the maximum slope for z .

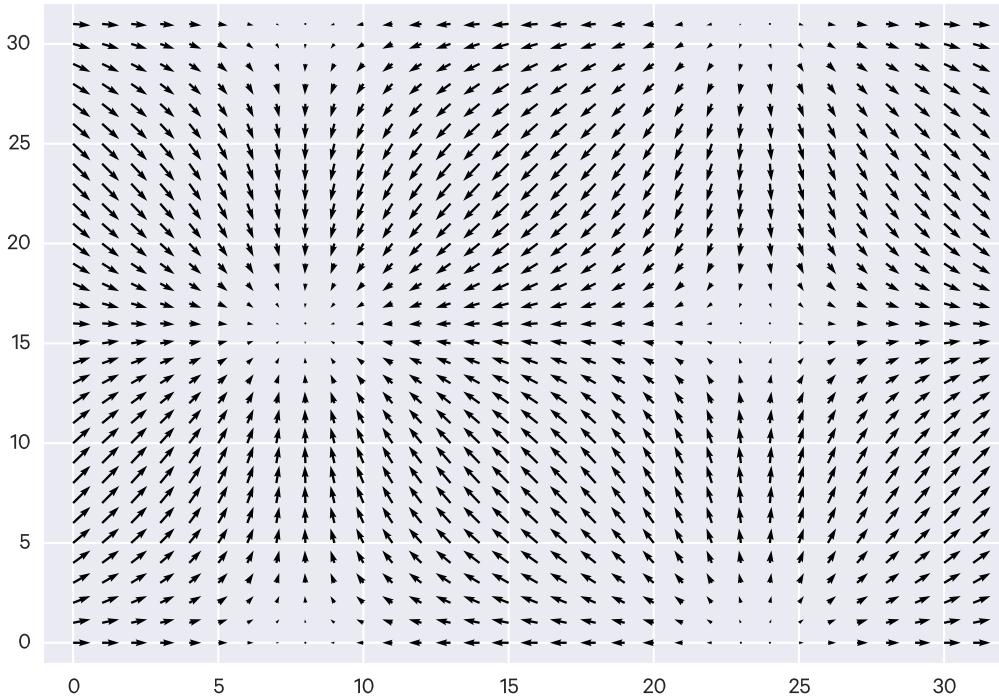
3.3.5 Scalar and vector fields

A **scalar field** just means a space where, for any point, you can get a scalar value.

For example, with $f(x, y) = x^2 + xy + y^2$, for any (x, y) you get a scalar value.

A **vector field** is similar but instead of just a scalar value, you get a value and a direction.

For example, $\vec{V} = 2x\hat{i} + 5y\hat{j}$ or $\vec{V} = x^2y\hat{i} + y\hat{j}$.



An example of a 2D vector field

3.3.6 Divergence

Say we have a vector field $\vec{V} = x^2y\hat{i} + 3y\hat{j}$.

The **divergence** of that vector field is:

$$\text{div}(\vec{V}) = \nabla \cdot \vec{V}$$

That is, it is the dot product (which tells us how much two vectors move together) of the gradient and the vector field.

So for our example:

$$\begin{aligned}\nabla &= \frac{\partial}{\partial x}\hat{i} + \frac{\partial}{\partial y}\hat{j} \\ \nabla \cdot \vec{v} &= \frac{\partial}{\partial x}(x^2y) + \frac{\partial}{\partial y}(3y) \\ &= 2xy + 3\end{aligned}$$

The divergence, which is scalar number for any point in a vector field, represents the change in volume density from an infinitesimal volume around a given point in that field. A positive divergence means the volume density is decreasing (more going out than coming in); a negative divergence means the

volume density is increasing (more is coming in than going on, this is also called **convergence**). A divergence of 0 means the volume density is not changing.

Using our previously calculated divergence, say we want to look at the point $(4, 3)$. We get the divergence $2 \cdot 4 \cdot 3 + 3 = 27$ This means that, in an infinitesimal volume around the point $(4, 3)$, the volume is decreasing.

3.3.7 Curl

The **curl** measures the rotational effect of a vector field at a given point. Unlike divergence, where we are seeing how much the gradient and the vector field move together, we are interested in seeing how they move against each other. So we use their cross product:

$$\text{curl}(\vec{V}) = \nabla \times \vec{V}$$

3.3.8 The Hessian

Say we have a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, which takes as input a some vector $x \in \mathbb{R}^n$ and returns a real number (that is, it defines a scalar field).

The **Hessian** matrix with respect to x , written $\nabla_x^2 f(x)$ or as just H , is the $n \times n$ matrix of second-order partial derivatives:

$$\nabla_x^2 f(x) \in R^{n \times n} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}$$

Which is to say $(\nabla_x^2 f(x))_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$.

The Hessian is always symmetric since $\frac{\partial^2 f(x)}{\partial x_i \partial x_j} = \frac{\partial^2 f(x)}{\partial x_j \partial x_i}$.

3.3.9 The Jacobian

For the vector $F(x) = [f(x)_1, \dots, f(x)_k]^T$, the **Jacobian**, notated $\nabla_x F(X)$ or as just J , is:

$$\nabla_x F(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x)_1 & \cdots & \frac{\partial}{\partial x_d} f(x)_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f(x)_k & \cdots & \frac{\partial}{\partial x_d} f(x)_k \end{bmatrix}$$

That is, it is an $m \times n$ matrix of the first-order partial derivatives for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, (i.e. for a function that defines a vector field).

The Jacobian and the Hessian are related by:

$$H(f)(x) = J(\nabla f)(x)$$

3.3.10 Optimization with eigenvalues

Consider, for a symmetric matrix $A \in \mathbb{S}^n$ the following equality-constrained optimization problem:

$$\max_{x \in \mathbb{R}^n} x^T A x, \text{ subject to } \|x\|_2^2 = 1$$

Optimization problems with equality constraints are typically solved by forming the **Lagrangian**, an objective function which includes the equality constraints. For this particular problem (i.e. with the quadratic form), the Lagrangian is:

$$\mathcal{L}(x, \lambda) = x^T A x - \lambda x^T x$$

Where λ is the **Lagrange multiplier** associated with the equality constraint. For x^* to be the optimal point to the problem, the gradient of the Lagrangian has to be zero at x^* (among other conditions), i.e.:

$$\nabla_x \mathcal{L}(x, \lambda) = \nabla_x (x^T A x - \lambda x^T x) = 2A^T x - 2\lambda x = 0$$

This is just the linear equation $Ax = \lambda x$, so the only points which can maximize (or minimize) $x^T A x$, assuming $x^T x = 1$, are the eigenvectors of A .

3.4 Differential Equations

Differential equations are simply just equations that contain derivatives.

Ordinary differential equations (ODEs) involve equations containing:

- variables
- functions
- their derivatives
- their solutions

This is contrasted to **partial differential equations** (PDEs), which contain partial derivatives instead of ordinary derivatives.

3.4.1 Solving simple differential equations

Say we have:

$$f''(x) = 2$$

First we can integrate both sides:

$$\begin{aligned}\int f''(x)dx &= \int 2dx \\ f'(x) &= 2x + C_1\end{aligned}$$

Then we can integrate once more:

$$\int f'(x)dx = \int 2x + C_1 dx \quad f(x) = x^2 + C_1 x + C_2$$

So our solution is $f(x) = x^2 + C_1 x + C_2$. For all values of C_1 and C_2 , we will get $f'' = 2$.

The values C_1 and C_2 represent *initial conditions*, e.g. the starting conditions of a model.

3.4.2 Basic first order differential equations

There are four main types (though there are many others) of differential equations:

- separable
- homogenous
- linear
- exact

Separable differential equations

A *separable equation* is in the form:

$$\frac{dy}{dx} = \frac{f(x)}{g(x)}$$

You can group the terms together like so:

$$g(y)dy = f(x)dx$$

And then integrate both sides to obtain the solution:

$$\int g(y)dy = \int f(x)dx + C$$

Example

Say we want to solve

$$\frac{dy}{dx} = 3x^2y$$

Separate the terms:

$$\frac{dy}{y} = (3x^2)dx$$

Then integrate:

$$\begin{aligned}\int \frac{dy}{y} &= \int 3x^2 dx \\ \ln y &= x^3 + C \\ y &= e^{x^3+C}\end{aligned}$$

If we let $k = e^C$ so k is a constant, we can write the solution as:

$$y = ke^{x^3}$$

Homogenous differential equations

A *homogenous equation* is in the form:

$$\frac{dy}{dx} = f(y/x)$$

To make things easier, we can use the substitution

$$v = \frac{y}{x}$$

so

$$\frac{dy}{dx} = f(v)$$

Then we can set $y = xv$ and use the product rule, so that we get:

$$\begin{aligned}\frac{dy}{dx} &= v + x\frac{dv}{dx} \\ v + x\frac{dv}{dx} &= f(v) \\ x\frac{dv}{dx} &= f(v) - v \\ \frac{dv}{dx} &= \frac{f(v) - v}{x}\end{aligned}$$

so now the equation is in separable form and be solved as a separable equation.

Linear differential equations

A *linear* first order differential equation is a differential equation in the form:

$$\frac{dy}{dx} + f(x)y = g(x)$$

To solve, you multiply both sides by $I = e^{\int f(x)dx}$ and integrate. I is known as the *integrating factor*.

Example

$$y' - 2xy = x$$

So in this case, $f(x) = -2x$, and $g(x) = x$, so the equation could be written:

$$y' + f(x)y = g(x)$$

So, we calculate the integrating factor:

$$I = e^{\int -2xdx} = e^{-x^2}$$

and multiply both sides by I , i.e.:

$$\begin{aligned}e^{-x^2}(y' - 2xy) &= xe^{-x^2} \\ (e^{-x^2} \cdot y') - 2xe^{-x^2}y &= xe^{-x^2} \\ \int((e^{-x^2} \cdot y') - 2xe^{-x^2}y)dx &= \int xe^{-x^2}dx\end{aligned}$$

and work out the integration.

Exact differential equations

An *exact equation* is in the form:

$$f(x, y) + g(x, y) \frac{dy}{dx} = 0$$

such that $\frac{df}{dx} = \frac{dg}{dy}$.

There exists some function $h(x, y)$ where

$$\begin{aligned}\frac{dh}{dx} &= f(x, y) \\ \frac{dh}{dy} &= g(x, y)\end{aligned}$$

so long as f , g , $\frac{df}{dy}$ and $\frac{dg}{dx}$ are continuous on a connected region.

3.5 References

- <https://en.wikibooks.org/wiki/Calculus>
- <https://www.khanacademy.org/math/multivariable-calculus/>
- Linear Algebra Review and Reference, Zico Kolter. October 16, 2007.

4

Probability

Probability theory is the study of uncertainty.

4.1 Probability space

We typically talk about the probability of an **event**. The **probability space** defines the possible outcomes for the event, and is defined by the triple (Ω, \mathcal{F}, P) , where

- Ω is the space of possible outcomes, i.e. the **outcome space** (sometimes called the **sample space**).
- $\mathcal{F} \subseteq 2^\Omega$, where 2^Ω is the power set of Ω (i.e. the set of all subsets of Ω , including the empty set \emptyset and Ω itself, the latter of which is called the **trivial event**), is the *space of measurable events* or the **event space**.
- P is the *probability measure*, i.e. the **probability distribution**, that maps an event $E \in \mathcal{F}$ to a real value between 0 and 1 (that is, P is a function that outputs a probability for the input event).

For example, we have a six-sided dice, so the space of possible outcomes $\Omega = \{1, 2, 3, 4, 5, 6\}$. We are interested in whether or not the dice roll is odd or even, so the event space is $\mathcal{F} = \{\emptyset, \{1, 3, 5\}, \{2, 4, 6\}, \Omega\}$.

The outcome space Ω may be finite, in which the event space \mathcal{F} is typically taken to be 2^Ω , or it may be infinite, in which the probability measure P must satisfy the following axioms:

- **non-negativity**: for all $\alpha \in \mathcal{F}$, $P(\alpha) \geq 0$.
- **trivial event**: $P(\Omega) = 1$.
- **additivity**: For all $\alpha, \beta \in \mathcal{F}$ and $\alpha \cap \beta = \emptyset$, $P(\alpha \cup \beta) = P(\alpha) + P(\beta)$.

Other axioms include:

- $0 \leq P(a) \leq 1$
- $P(\text{True}) = 1$ and $P(\text{False}) = 0$

We refer to an event whose outcome is unknown as a *trial*, or an *experiment*, or an *observation*. An event is a trial which has resolved (we know the outcome), and we say “the event has occurred” or that the trial has “satisfied the event”.

The **compliment** of an event is everything in the outcome space that is *not* the event, and may be notated in a few ways: $\neg E$, E^C , \bar{E} .

If two events cannot occur together, they are **mutually exclusive**.

4.2 Random Variables

A **random variable** (sometimes called a **stochastic variable**) is a function which maps outcomes to real values (that is, they are technically not variables but rather functions), dependent on some other probabilistic factor. Random variables represent uncertain events we are interested in with a numerical value.

Random variables are typically denoted by a capital letter, e.g. X . Values they may take are typically represented with a lowercase letter, e.g. x .

When we use $P(X)$ we are referring to the **distribution** of the random variable X , which describes the probabilities that X takes on each of its possible values.

Contrast this to $P(x)$ which describes the *probability* of some arbitrary single value x ; this is shorthand for describing the probability that some random variable (e.g. X) takes on that value x , which is more explicitly denoted $P(X = x)$ or $P_X(x)$. This represents some real value.

For example, we may be flipping a coin and have a random variable X which takes on the value 0 if the flip results in heads and 1 if it results in tails. If it's a fair coin, then we could say that $P(X = \text{heads}) = P(X = \text{tails}) = 0.5$.

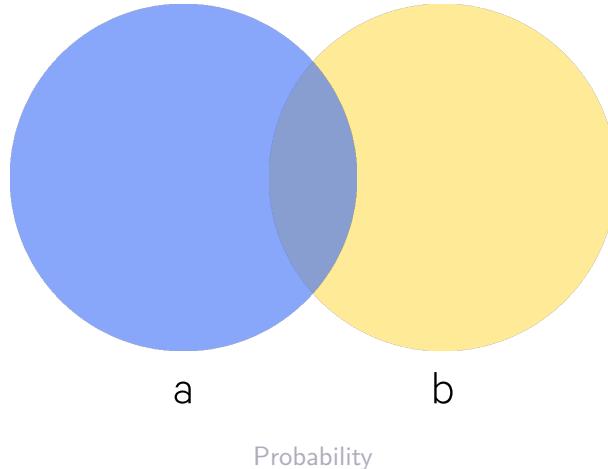
Random variables may be:

- **Discrete**: the variable can only have specific values, e.g. on a 5 star rating system, the random variable could only be one of the values $[0, 1, 2, 3, 4, 5]$. Another way of describing this is that the space of the variable's possible values (i.e. the outcome space) is *countable* and finite. For discrete random variables which are not numeric, e.g. gender (male, female, etc), we use an *indicator function* / to map non-numeric values to numbers, e.g. male = 0, female = 1, . . . ; we call variables from such functions **indicator variables**.
- **Continuous**: the variable can have arbitrarily exact values, e.g. time, speed, distance. That is, the outcome space is infinite.
- **Mixed**: these variables assign probabilities to both discrete and continuous random variables.

4.3 Joint and disjoint probabilities

- **Joint probability:** $P(a \cup b) = P(a \wedge b) = P(a, b) = P(a)P(b|a)$, the probability of both a and b occurring.
- **Disjoint probability:** $P(a \cap b) = P(a \vee b) = P(a) + P(b) - P(a, b)$, the probability of a or b occurring.

Probabilities can be visualized as a Venn diagram:



The overlap is where both a and b occur ($P(a, b)$).

The previous axiom, describing $P(a \cap b)$, adds the spaces where a and b each occur, but this counts their overlap twice, so we subtract it once.

4.4 Conditional Probability

The conditional probability is the probability of A given B , notated $P(A|B)$.

Formally, this is:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

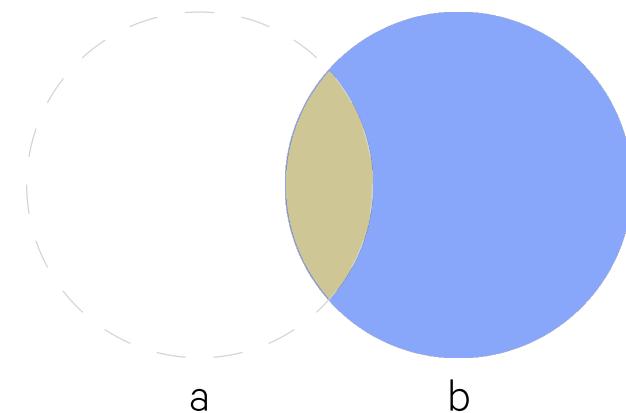
Where $P(B) > 0$.

For example, say you have two die. Say $A = \{\text{snake eyes}\}$ and $B = \{\text{double}\}$.

$P(A) = \frac{1}{36}$ since out of the 36 possible dice pairings, only one is snake eyes. $P(B) = \frac{1}{6}$ since 6 of the 36 possible dice pairings are doubles.

Now what is $P(A|B)$? Well, intuitively, if B has happened, we have reduced our possible event space to just the 6 doubles, one of which is snake eyes, so $P(A|B) = \frac{1}{6}$.

Intuitively, this can be thought of as the probability of a given a universe where b occurs.



The overlapping region is $P(a|b)$

So we ignore the part of the world in which b does not occur.

This can be re-written as:

$$P(a, b) = P(a|b)P(b)$$

4.5 Independence

Events X and Y are independent if:

$$P(X \cap Y) = P(X)P(Y)$$

That is, their outcomes are unrelated.

Another way of saying this is that:

$$P(X|Y) = P(X)$$

Knowing something about Y tells us nothing more about X .

The independence of X and Y can also be notated as $X \perp Y$.

From this we can infer that:

$$P(X, Y) = P(X)P(Y)$$

More generally we can say that events A_1, \dots, A_n are **mutually independent** if $P(\bigcap_{i \in S} A_i) = \prod_{i \in S} P(A_i)$ for any $S \subset \{1, \dots, n\}$. That is, the joint probability of any subset of these events is just equal to the product of their individual probabilities.

Mutual independence implies pairwise independence, but note that the converse is not true (that is, pairwise independence does not imply mutual independence).

4.5.1 Conditional Independence

Conditional independence is defined as:

$$P(X|Y, Z) = P(X|Z)$$

If X is independent of Y conditioned on Z . Which is to say X is independent of Y if Z is true or known.

From this we can infer that:

$$P(X, Y|Z) = P(X|Z)P(Y|Z)$$

Note that mutual independence does not imply conditional independence.

Similarly, we can say that events A_1, \dots, A_n are conditionally independent given C if $P(\bigcap_{i \in S} A_i | C) = \prod_{i \in S} P(A_i | C)$ for any $S \subset \{1, \dots, n\}$.

4.6 The Chain Rule of Probability

Say we have the joint probability $P(a, b, c)$. How do we turn this into conditional probabilities?

We can set $y = b, c$ (that is, the intersection of b and c), then we have $P(a, b, c) = P(a, y)$, and we can just apply the previous equation:

$$\begin{aligned} P(a, y) &= P(a|y)P(y) \\ P(a, b, c) &= P(a|b, c)P(b, c) \end{aligned}$$

And we can again apply the previous equation to $P(b, c)$, which gets us:

$$P(a, b, c) = P(a|b, c)P(b|c)P(c)$$

This is generalized as the **chain rule of probability**:

$$P(x_1, \dots, x_n) = \prod_{i=n}^1 P(x_i|x_{i-1}, \dots, x_1)$$

4.7 Combinations and Permutations

4.7.1 Permutations

With permutations, *order matters*. For instance, AB and BA are different *permutations* (though they are the same *combination*, see below).

Permutations are notated:

$${}_x P_y = P_y^x = P(x, y)$$

Where:

- x = total number of “items”
- y = “spots” or “spaces” or “positions” available for the items.

A permutation can be expanded like so:

$${}_n P_k = n, n - 1, n - 2, \dots, n - (k - 1)$$

And generalized to the following formula:

$${}_n P_k = \frac{n!}{(n - k)!}$$

For example, consider ${}_7 P_3$. $7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$, and we only have 3 positions, i.e. 7-6-5, so we divide by $4!$ to get our final answer.

4.7.2 Combinations

With combinations, the order *doesn't matter*.

The notation is basically the same as permutation, except with a C instead of a P :

$${}_n C_k = \frac{{}_n P_k}{k!}$$

or, expanded:

$${}_n C_k = \frac{n!}{k!(n - k)!} = \binom{n}{k}$$

The n and k pairing together is known as the **binomial coefficient**, and read as “ n choose k ”.

4.7.3 Combinations, permutations, and probability

Example

Say you have a coin. What is $P(\frac{3}{8}H)$? That is, what is the probability of flipping exactly 3 heads?

$P(\frac{3}{8}H)$ is equal to the combination of $\binom{8}{3}$. That is, we are basically trying to find all the combinations of 3 head flips out of the total 8 flips.

$${}_8C_3 = 56$$

So there are 56 possible outcomes that result in exactly 3 heads. Because a coin has two possible outcomes, and we're flipping 8 times, we know there are 2^8 total possible outcomes.

So to figure out the probability, we can just take the ratio of these outcomes.

$$P(\frac{3}{8}H) = \frac{56}{2^8} = \frac{7}{32}$$

Example

Given outcome $P(A) = 0.8$ and $P(B) = 0.2$, what is $P(\frac{3}{5}A)$? That is, the possibility of exactly 3 out of 5 trials being A.

Basically, like before, we're looking for possible combinations of $\frac{3}{5}A$, that is 5 choose 3, i.e. ${}_5C_3 = 10$.

So we know there are 10 possible outcomes resulting in $\frac{3}{5}A$. But what is the probability of a single combination that results in $\frac{3}{5}A$? We were given the probabilities so it's just multiplying:

$$P(A)P(A)P(A)P(B)P(B) = 0.8^3 \times 0.2^2$$

So then we just multiply the number of these combinations, 10, by this resulting probability to get the final answer.

4.8 Probability Distributions

For some random variable X , there is a **probability distribution function** $P(X)$ (usually just called the **probability distribution**); the particular kind depends on what kind of random variable X is (i.e. discrete or continuous). A probability distribution describes the probability of a random variable taking on its possible values.

If the random variable X is distributed according to, say, a Poisson distribution, we say that “ X is Poisson-distributed”.

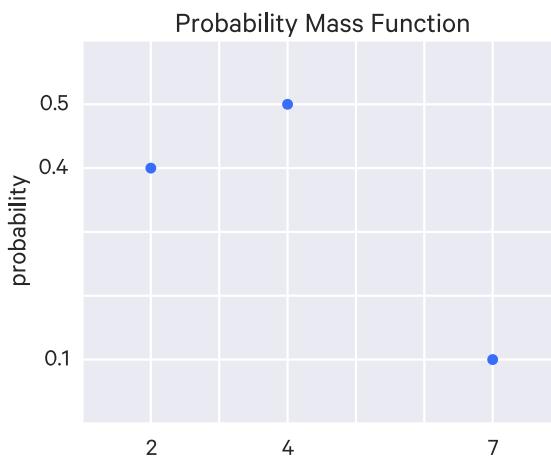
Distributions themselves are described by **parameters** - variables which determine the specifics of a distribution. Different kinds of distributions are described by different sets of parameters. For instance, the particular shape of a normal distribution is determined by μ (the mean) and σ (the standard deviation); we say the normal distribution is **parameterized** by μ and σ . Often we are given a set of data and assume it comes from a particular kind of distribution, such as a normal distribution, but we don’t know the specific *parameterization* of the distribution; that is, with the normal distribution example, we don’t know what μ and σ are, so we use the data we have and try and infer these unknown parameters.

4.8.1 Probability Mass Functions (PMF)

For discrete random variables, the distribution is a **probability mass function**.

It is called a “mass function” because it divides a unit mass (the total probability) across the different values the random variable can take.

In the example figure, the random variable can take on one of three discrete values, $\{1, 3, 7\}$, with the corresponding probabilities illustrated in the PMF.



An example PMF

4.8.2 Probability Density Functions (PDF)

For continuous random variables we have a **probability density function**. A probability density function f is a non-negative, integrable function such that:

$$\int_{\text{Val}(X)} f(x) dx = 1$$

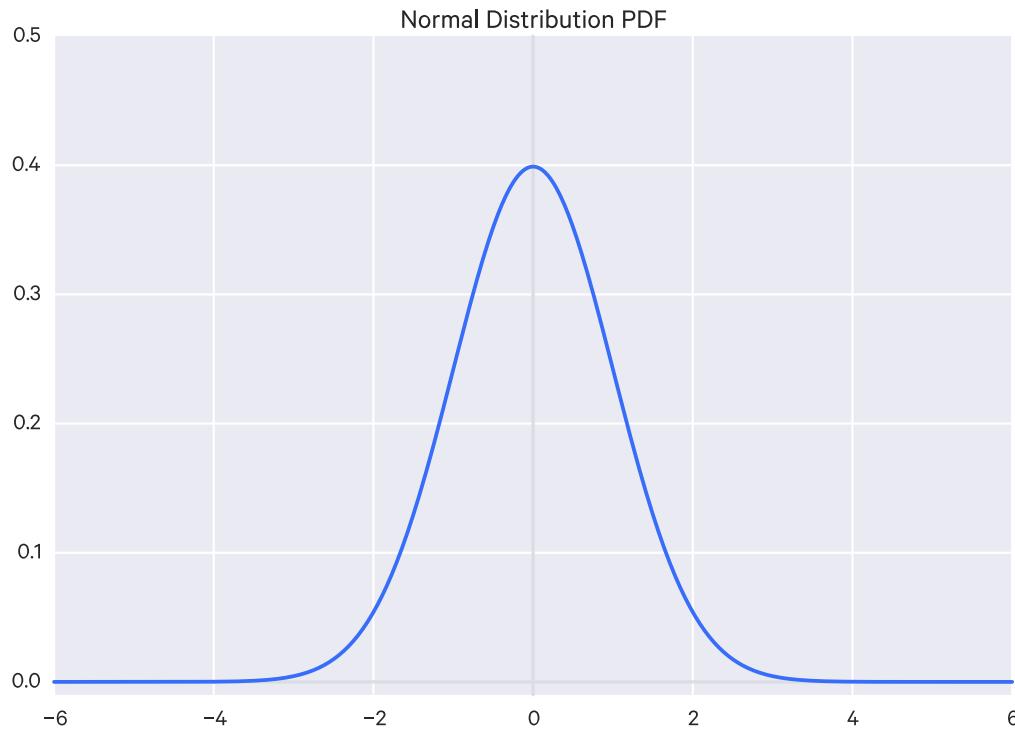
Where $\text{Val}(X)$ denotes the range of the random variable X , so this is the integral over the range of possible values for X .

The total area under the curve sums to 1 (which is to say that the aggregate probability of all possible values for X sums to 1).

The probability of a random variable X distributed according to a PDF f is computed:

$$P(a \leq X \leq b) = \int_a^b f(x)dx$$

It's worth noting that this implies that, for a continuous random variable X , the probability of taking on any given *single* value is zero (when dealing with continuous random variables there are infinitely precise values). Rather, we compute probabilities for a range of values of X ; the probability is given by the area under the curve over this range.



An example PDF (the normal distribution)

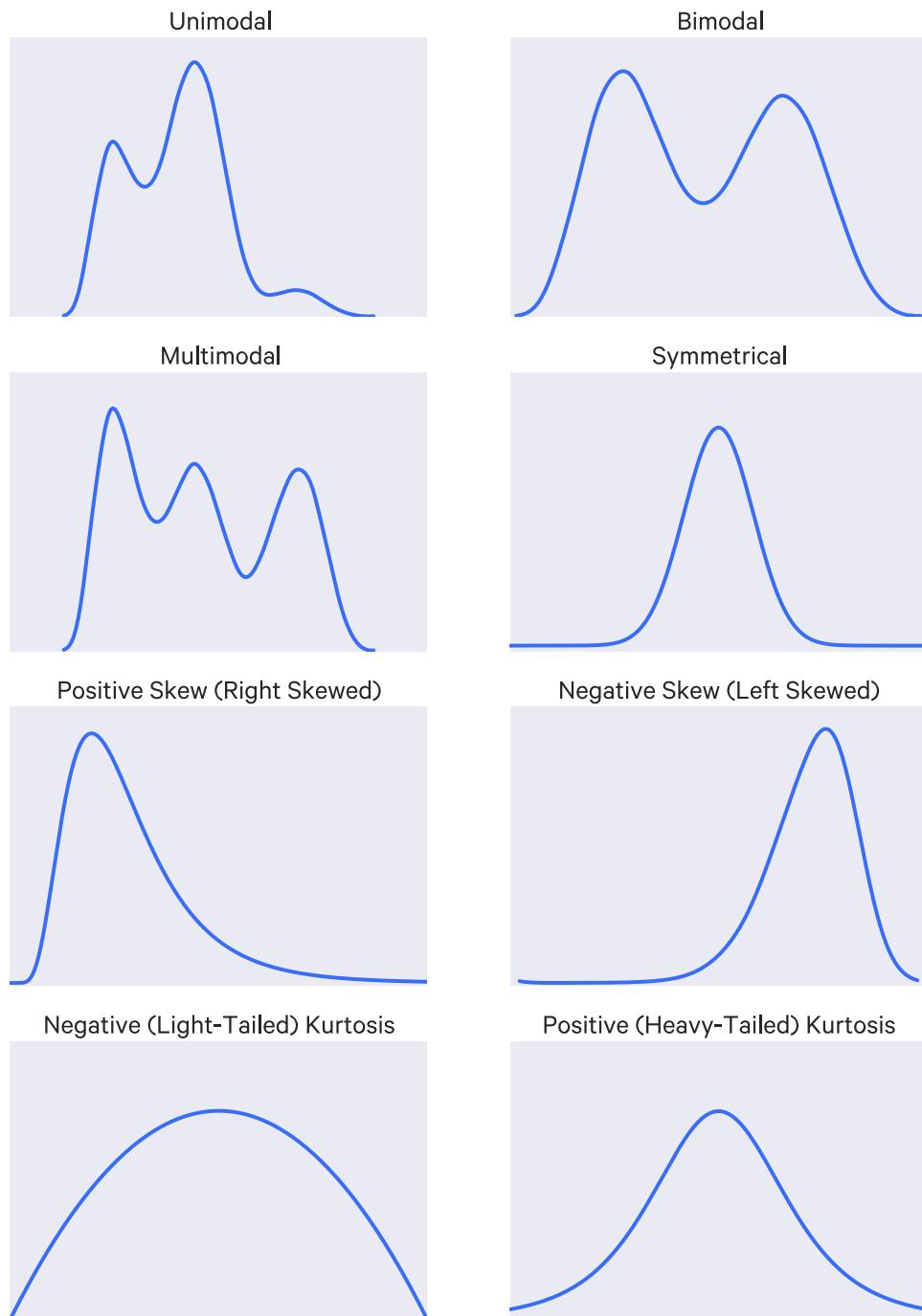
4.8.3 Distribution Patterns

There are a few ways we can describe distributions.

- **Unimodal:** The distribution has one main peak.
- **Bimodal:** The distribution has two (approximately) equivalent main peaks.
- **Multimodal:** The distribution has more than two (approximately) equivalent main peaks.
- **Symmetrical:** The distribution falls in equal numbers on both sides of the middle.

Skewness

Skewness describes distributions that have greater density on one side of the distribution. The side with less is the direction of the skew.



Common distribution descriptors

Skewness is defined:

$$\text{skewness} = E\left[\left(\frac{X - \mu}{\sigma}\right)^3\right]$$

Where σ is just the standard deviation.

The normal distribution has a skewness of 0.

Kurtosis

Kurtosis describes how the shape differs from a normal curve (if the tails are lighter or heavier).

Kurtosis is defined:

$$\text{kurtosis} = \frac{E[(X - \mu)^4]}{(E[(X - \mu)^2])^2}$$

The standard normal distribution has a kurtosis of 3, so sometimes kurtosis is standardized by subtracting 3; this standardized kurtosis measure is called the *excess kurtosis*.

4.9 Cumulative Distribution Functions (CDF)

A cumulative distribution function $\text{CDF}(x)$, often also denoted $F(x)$, describes the cumulative probability up to where the random variable takes the value x , which is to say it tells you $P(X \leq x)$.

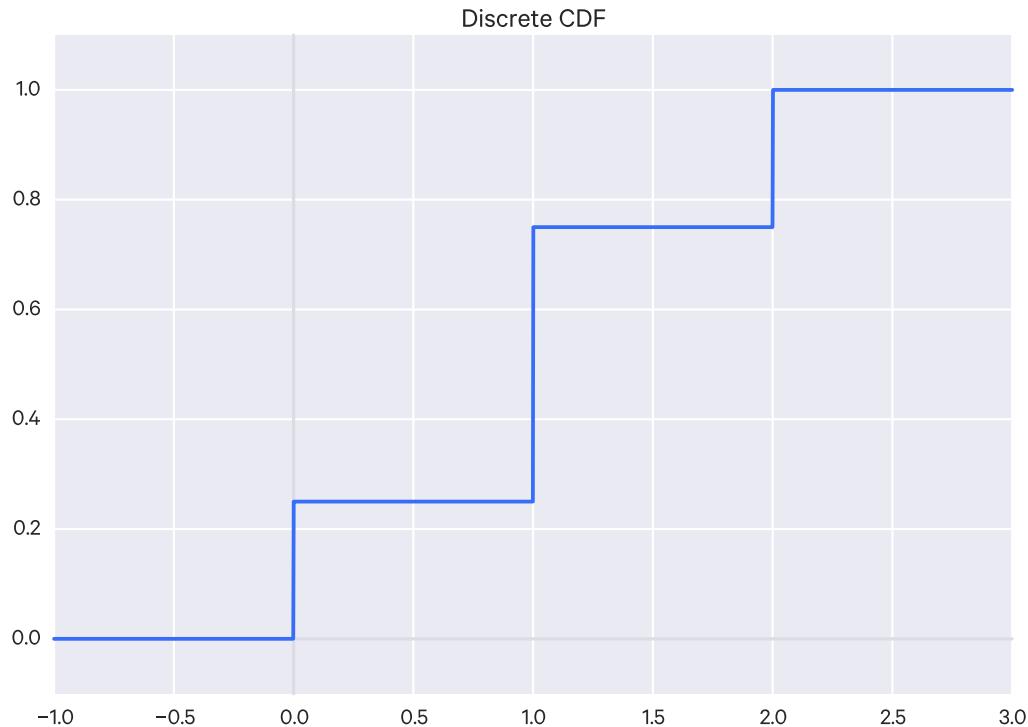
4.9.1 Discrete random variables

The cumulative distribution function of a discrete random variable is just the sum of the probabilities for the values up to x .

Say our discrete random variable X can take values from $\{x_1, \dots, x_n\}$. We can define the CDF for X as:

$$\text{CDF}(x) = \sum_{\{x_i | x_i \leq x\}} P(X = x_i)$$

The complete discrete CDF is a step function, as you might expect because the CDF is constant between discrete values.



An example discrete CDF

4.9.2 Continuous random variables

The cumulative distribution function of a continuous random variable is:

$$\text{CDF}(x) = \int_{-\infty}^x P(X)dx$$

That is, it is the integral of the PDF up to the value in question.

Probability values for a specific range $x_1 \rightarrow x_2$ can be calculated with:

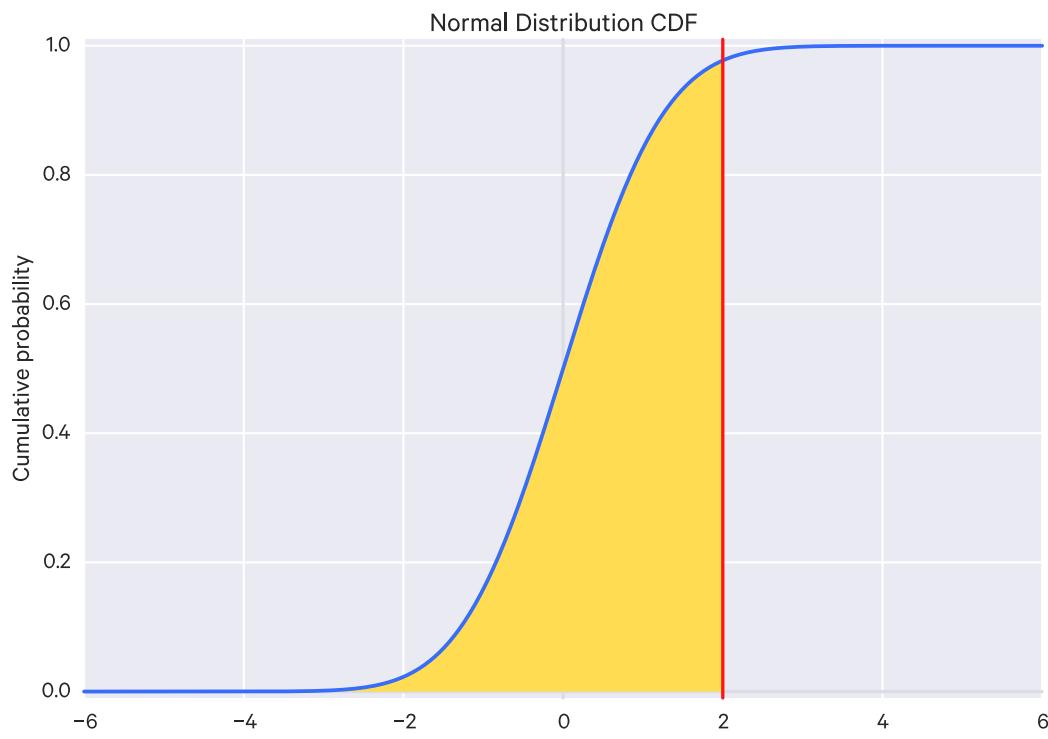
$$\text{CDF}(x_2) - \text{CDF}(x_1)$$

or more simply:

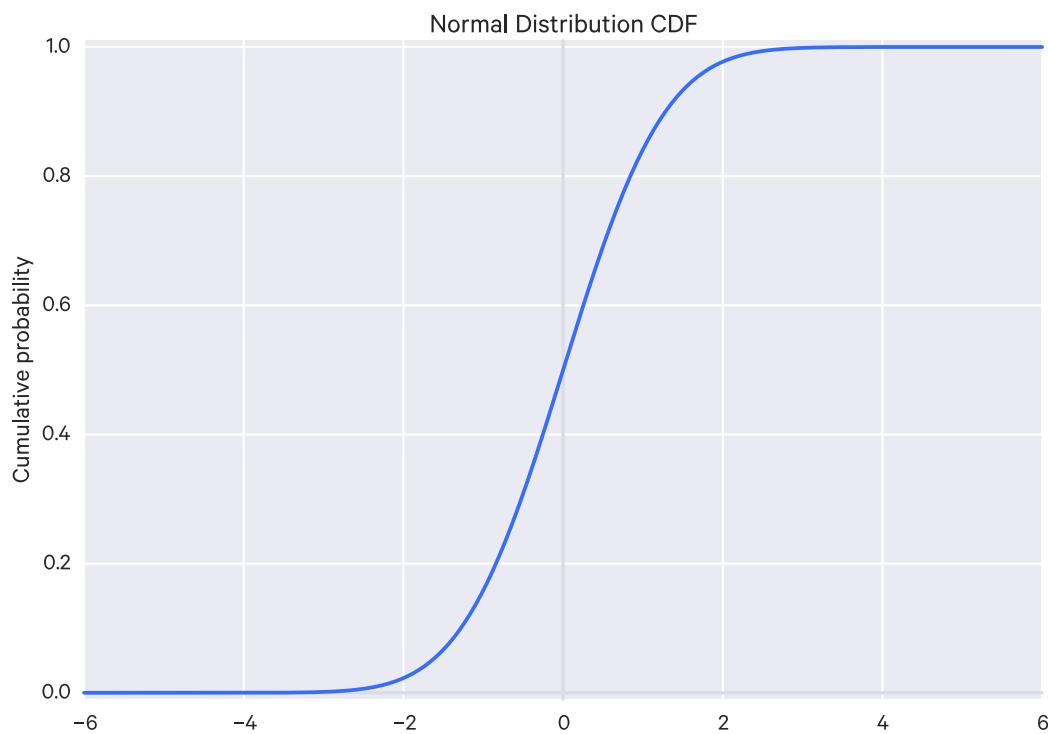
$$\text{CDF}(x) = \int_{x_1}^{x_2} P(x)dx$$

4.9.3 Using CDFs

Visually, there are a few tricks you can do with CDFs.

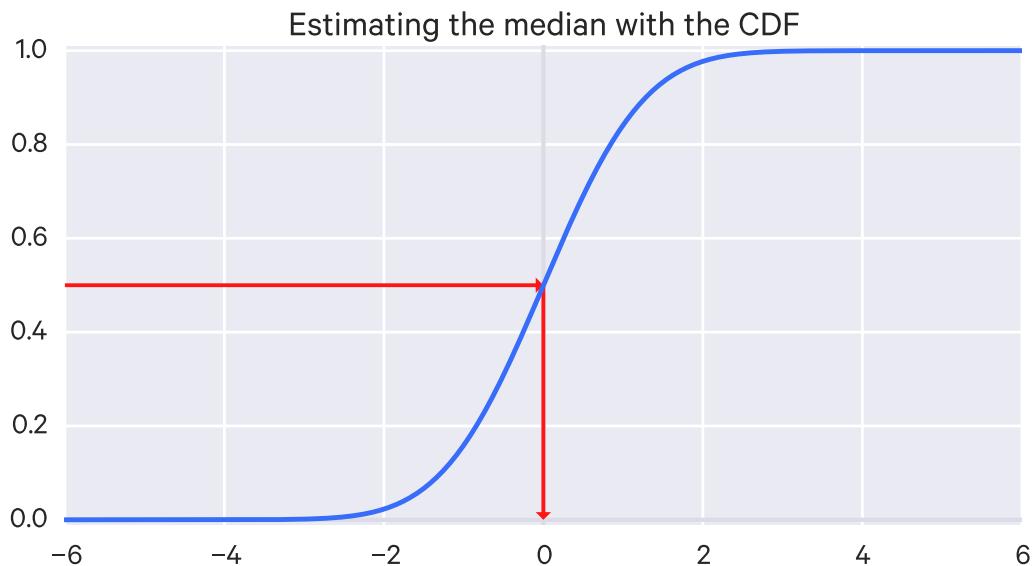


CDF



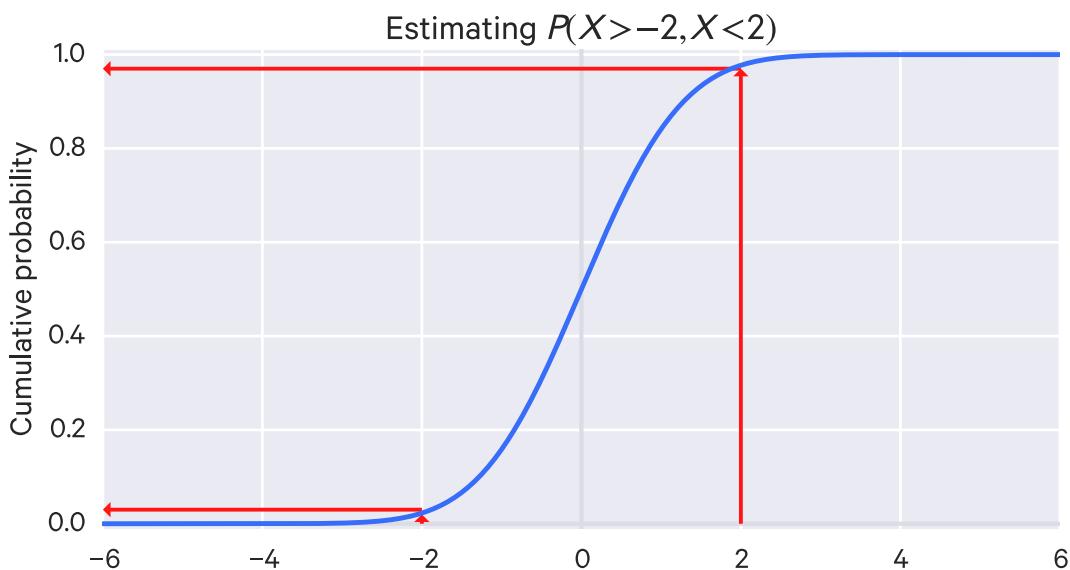
An example continuous CDF (for the normal distribution)

You can estimate the median by looking at where $CDF(x) = 0.5$, i.e.:



Estimating the median value with a CDF

You can estimate the probability that your x falls between two values:

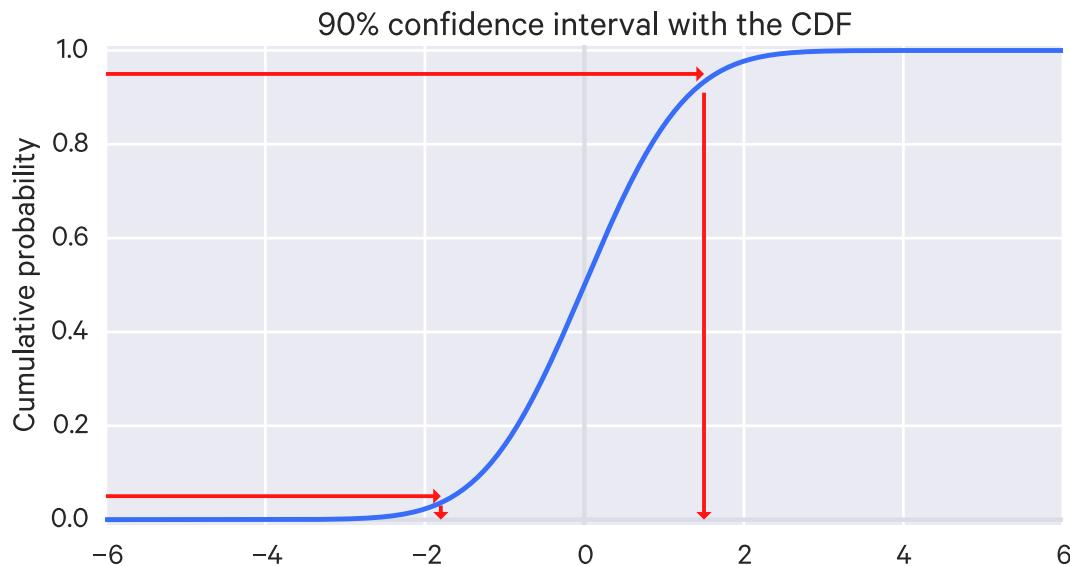


Estimating the probability of value falling in a range with a CDF

You can estimate a confidence interval as well. For example, the 90% confidence interval by looking at the x values in the range which produces $CDF(x) = 0.15$ and $CDF(x) = 0.95$.

4.9.4 Survival function

The **survival function** of a random variable X is the complement of the CDF, that is, it is the probability that the random variable is greater than some value x , i.e. $P(X > x)$. So the survival function is:



Estimating the 80% confidence interval with a CDF

$$S(x) = P(X > x) = 1 - \text{CDF}(x)$$

4.10 Expected Values

The expected value of a random variable X is:

$$E[X] = \mu$$

That is, it is the average (mean) value.

It can be thought of as a sample from a potentially infinite population. A sample from that population is expected to be the mean of that population. The value of that mean depends on the distribution of that population.

4.10.1 Discrete random variables

For a discrete random variable X with a sample space $X(\Omega)$ (i.e. all possible values X can take) and with a PMF p , the expected value is:

$$E[X] = \sum_{x \in X(\Omega)} xP(X = x)$$

The expected value exists only if this sum is well-defined, which basically means it has to aggregate in some clear way, as either a finite value or positive or negative infinity. But it can't, for instance,

contain a positive infinity and a negative infinity term simultaneously, because it's undefined how those combine.

For example, consider the infinite sum $1 - 1 + 1 - 1 + \dots$. The -1 terms go to negative infinity and the $+1$ terms go to positive infinity - this is an undefined sum.

4.10.2 Continuous random variables

For a continuous random variable X and a PDF f , the expected value is:

$$E[X] = \int_{-\infty}^{\infty} xf(x)dx$$

The expected value exists only when this integral is well-defined.

4.10.3 The expectation rule

A function $g(X)$ of a random variable X is itself a random variable. The expected value of that function can be expressed based on the random variable X , like so:

$$\begin{aligned} E[g(x)] &= \sum_{x \in X(\Omega)} g(x)p(x) \\ E[g(x)] &= \int_{-\infty}^{\infty} g(x)f(x)dx \end{aligned}$$

Using whichever is appropriate, depending on if X is discrete or continuous.

4.10.4 Properties of expectations

For random variables X and Y where $E[|X|] < \infty$ and $E[|Y|] < \infty$ (that is, $E[X], E[Y]$ are finite), we have the following properties:

1. $E(a) = a$ for all $a \in \mathbb{R}$. That is, the expected value of a constant is just the constant. This is called the *normalization* property.
2. $E(aX) = aE(X)$ for all $a \in \mathbb{R}$
3. $E(X + Y) = E(X) + E(Y)$
4. If $X \geq 0$, that is, all possible values of X are greater than 0, then $E[X] \geq 0$
5. If $X \leq Y$, that is, each possible value of X is less than each possible value of Y , then $E[X] \leq E[Y]$. This is called the *order* property.
6. If X and Y are independent, then $E[XY] = E[X]E[Y]$. Note that the converse is not true; that is, if $E[XY] = E[X]E[Y]$, this does not necessarily mean that X and Y are independent.
7. $E[I_A(X)] = P(X \in A)$, that is, the expected value of an indicator function:

$$I_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases}$$

is the probability that the random variable X is in A .

Properties 2 and 3 are called *linearity*.

To put linearity another way: Let X_1, X_2, \dots, X_n be random variables, which may be dependent or independent:

$$E(X_1 + X_2 + \dots + X_n) = E(X_1) + E(X_2) + \dots + E(X_n)$$

4.11 Variance

The **variance** of a distribution is the “spread” of a distribution.

The variance of a random variable X tells us how spread out the data is along that variable’s axis/dimension.

It can be defined in a couple ways:

$$\text{Var}(X) = E[(X - E[X])^2] = E[X^2] - E[X]^2$$

Variance is not a linear function of X , for instance:

$$\text{Var}(aX + b) = a^2 \text{Var}(X)$$

If random variables X and Y are independent, then:

$$\text{Var}(X + Y) = \text{Var}(X) \text{Var}(Y)$$

4.11.1 Covariance

The **covariance** of two random variables is a measure of how “closely related” they are:

$$\text{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])]$$

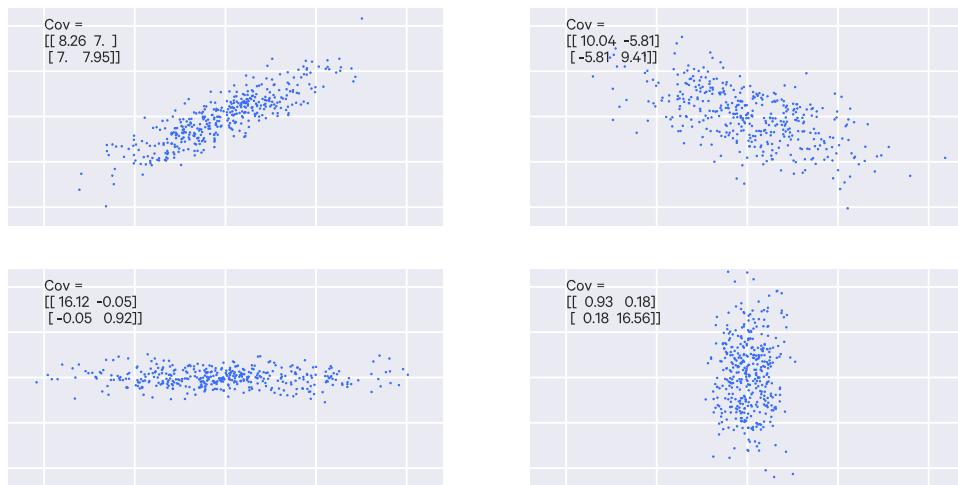
With more than two variables, a **covariance matrix** is used.

Covariance matrices show two things:

- the variance of a variable i , located at the i, i element
- the covariance of variables i, j , located at the i, j and j, i elements

If the covariance between two variables is negative, then we have a downward slope, if it is positive, then we have an upward slope.

So the covariance matrix tells us a lot about the shape of the data.



Example covariances

4.12 Common Probability Distributions

Here a few distributions you are likely to encounter are described in more detail.

4.12.1 Probability mass functions

Bernoulli Distribution

A random variable distributed according to the Bernoulli distribution can take on two possible values

0, 1, typically described as “failure” and “success” respectively. It has one parameter p , the probability of success, which is taken to be $P(X = 1)$. Such a random variable is sometimes called a **Bernoulli random variable**.

The distribution is described as:

$$P(X) = p^x(1 - p)^{1-x}$$

And for a Bernoulli random variable X , it is notated $X \sim \text{Ber}(p)$). X is 1 with probability p and X is 0 with probability $1 - p$.

The mean of a Bernoulli distribution is $\mu = p$, and the standard deviation is $\sigma = \sqrt{p(1 - p)}$.

A Bernoulli distribution describes a single trial, though often you may consider multiple trials, each with its own random variable.

Geometric Distribution

Say we have a set of iid Bernoulli random variables, each representing a trial. What is the probably of finding the first success at the n th trial?

This can be described with a geometric distribution, which is a distribution where the probabilities decrease exponentially fast.

It is formalized as:

$$P(n) = (1 - p)^{n-1}p$$

With the mean $\mu = \frac{1}{p}$ and the standard deviation $\sigma = \sqrt{\frac{1-p}{p^2}}$.

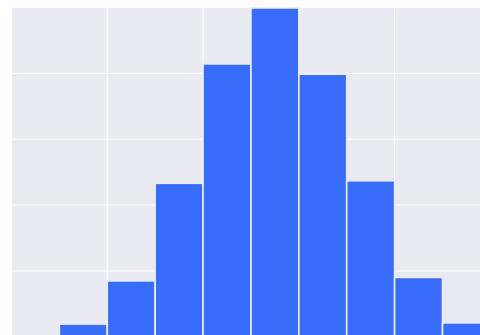
Binomial Distribution

Suppose you have a binomial experiment (i.e. one with two mutually exclusive outcomes, such as “success” or “failure”) of n trials (that is, n Bernoulli trials), where p is the probability of success on an individual trial, and is the same across all trials (that is, the trials are n iid Bernoulli random variables). You want to determine the probability of k successes in those n trials.

Note that *binomial* is in contrast to *multinomial* in which a random variable can take on more than just two discrete values. This shouldn't be confused with *multivariate* which refers to a situation where there are multiple variables.

The resulting distribution is a **binomial distribution**, such as:

The binomial distribution has the following properties:



A Binomial Distribution histogram

$$\mu = np$$

$$\sigma = \sqrt{np(1 - p)}$$

The binomial distribution is expressed as:

$$P(X = k) = \binom{n}{k} p(1 - p)^{n-k}$$

A binomial random variable X is denoted:

$$X \sim \text{Bin}(n, p)$$

Here X ends up being the number of events that occurred over our trials.

Its expected value is:

$$E[Z|N, p] = Np$$

The binomial distribution has two parameters:

- n - a positive integer representing the number of trials
- p - the probability of an event occurring in a single trial

The special case $N = 1$ corresponds to the *Bernoulli distribution*.

If we have Z_1, Z_2, \dots, Z_N Bernoulli random variables with the same p , then $X = Z_1 + Z_2 + \dots + Z_N \sim \text{Bin}(N, p)$.

Thus the expected value of a Bernoulli random variable is p (because $N = 1$).

Some example questions that can be answered with a binomial distribution:

- Out of ten tosses, how many times will this coin be heads?
- From the children born in a given hospital on a given day, how many of them will be girls?
- How many students in a given class room will have green eyes?
- How many mosquitoes, out of a swarm, will die when sprayed with insecticide?

([Source](#))

When the number of trials n gets large, the shape of the binomial distribution starts to approximate a normal distribution with the parameters $\mu = np$ and $\sigma = \sqrt{np(1 - p)}$.

Negative Binomial Distribution

The negative binomial distribution is a more general form of the geometric distribution; instead of giving the probability of the *first* success in the n th trial, it gives the probability of an arbitrary k th success in the n th trial. Like the geometric and binomial distribution, it is expected that the trials are iid Bernoulli random variables. The other requirement is that the last trial is a success.

This distribution is described as:

$$P(k|n) = \binom{n-1}{k-1} p^k (1-p)^{n-k}$$

Poisson Distribution

The **Poisson distribution** is useful for describing the number of rare (independent) events in a large population (of independent individuals) during some time span. It looks at how many times a discrete event occurs, over a period of continuous space or time; without a fixed number of trials.

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}, k = 0, 1, 2, \dots$$

If X is Poisson-distributed, we denote it:

$$X \sim \text{Poisson}(\lambda)$$

For the Poisson distribution λ is any positive integer. Its size is proportional to the probability of larger values in the distribution. That is, increasing λ assigns more probability to large values; decreasing it assigns more probability to small values. It is sometimes called the *intensity* of the distribution.

For the Poisson distribution, λ is known as the “(average) arrival rate” or sometimes just the “rate”.

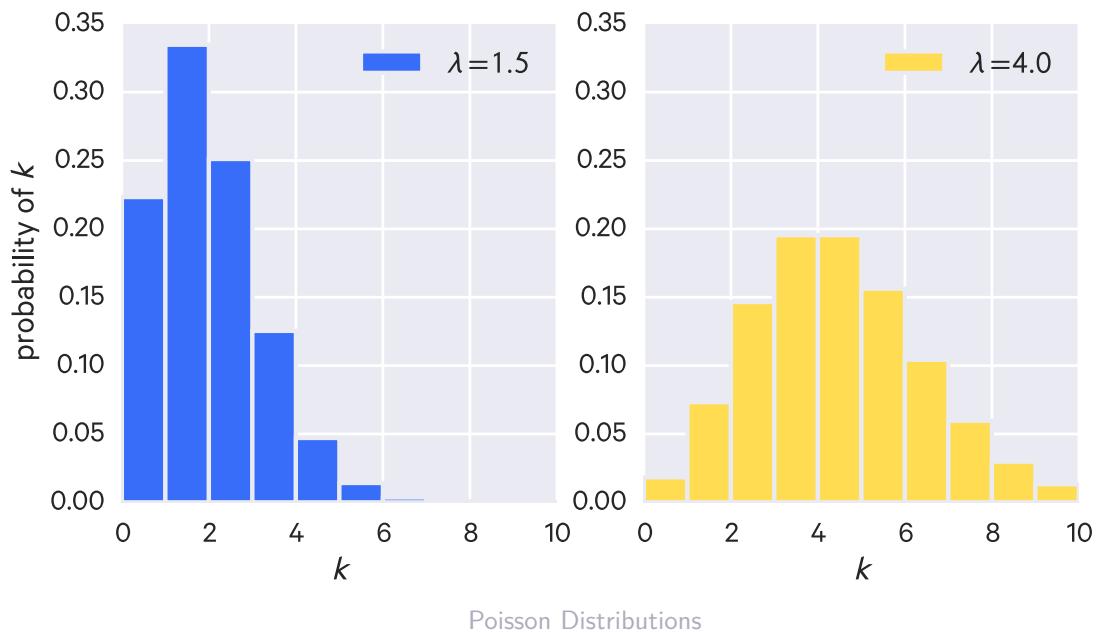
k must be a non-negative integer (e.g. 0, 1, 2, …).

A shorthand for saying that X has a Poisson mass distribution is $X \sim \text{Poisson}(\lambda)$.

For Poisson distributions, the expected value of our random variable is equal to the parameter λ , that is:

$$E[X|\lambda] = \mu = \lambda$$

In the Poisson distribution figure, although it looks like the values fall off at some point, it actually has an infinite tail, so that every positive integer has some positive probability.



Example

On average, 9 cars pass this intersection every hour. What is the probability that two cars pass the intersection this hour? Assume a Poisson distribution.

This problem can be framed as: what is $P(x = 2)$?

We know the expected value is 9 and that we have a Poisson distribution, so $\lambda = 9$ and:

$$P(x = 2) = \frac{9^2}{2!} e^{-9}$$

Some example questions that can be answered with a Poisson distribution:

- How many pennies will I encounter on my walk home?
- How many children will be delivered at the hospital today?
- How many products will I sell after airing a new television commercial?
- How many mosquito bites did you get today after having sprayed with insecticide?
- How many defects will there be per 100 metres of rope sold?

([Source](#))

4.12.2 Probability density functions

Uniform distribution

With the uniform distribution, every value is equally likely.

It may be constrained to a range of values as well.

Exponential Distribution

A random variable which is continuous may have an *exponential density*, often described as an *exponential random variable*:

$$f_X(x|\lambda) = \lambda e^{-\lambda x}, x \geq 0$$

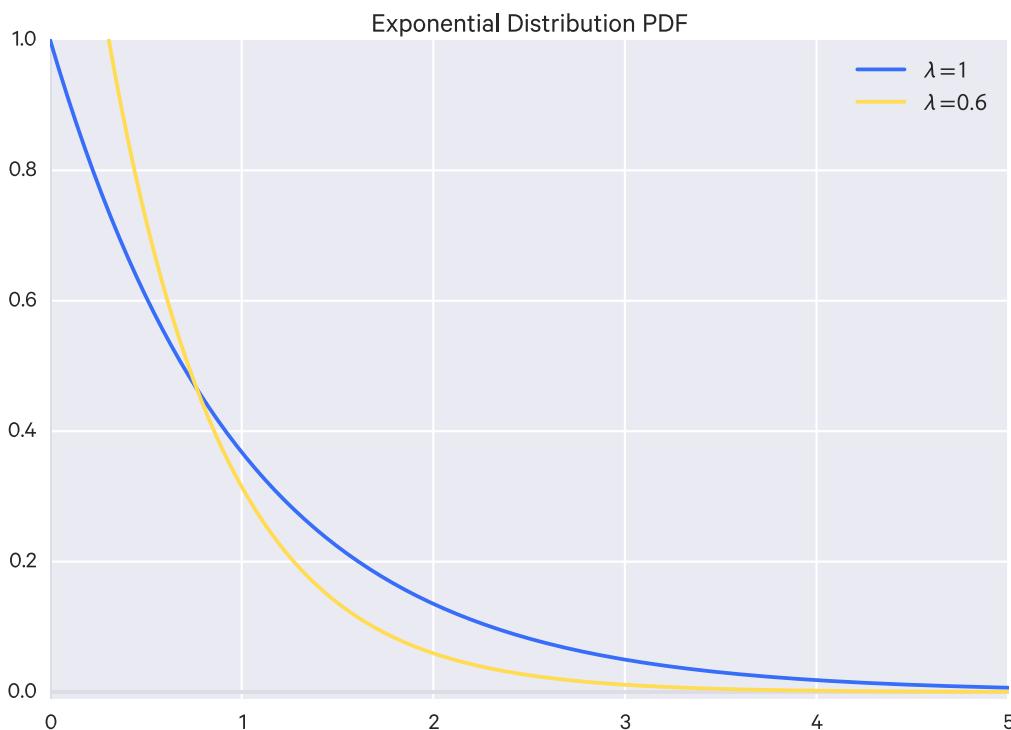
Here we say X is *exponential*:

$$X \sim \text{Exp}(\lambda)$$

Like the Poisson random variable, the exponential random variable can only have positive values. But because it is continuous, it can also take on non-integral values such as 4.25.

For exponential distributions, the expected value of our random variable is equal to the inverse of the parameter λ , that is:

$$E[X|\lambda] = \frac{1}{\lambda}$$



An Exponential Distribution

Example

Say we have the random variable y which is the exact amount of rain we will get tomorrow, in inches. What is the probability that $y = 2 \pm 0.1$? Assume you have the probability density function f for y .

We'd note the probability we're looking for like so:

$$P(|Y - 2| < 0.1)$$

Which is the probability that $y \approx 2$ within a tolerance (acceptable deviance) of 0.1 (i.e. 1.9 to 2.1).

Then we would just find the integral (area under the curve) of the PDF from 1.9 to 2.1, i.e.

$$\int_{1.9}^{2.1} f(x) dx$$

Gamma Distribution

$$X \sim \text{Gam}(\alpha, \beta)$$

This is over positive real numbers.

It is just a generalization of the exponential random variable:

$$\text{Exp}(\beta) \sim \text{Gam}(1, \beta)$$

The PDF is:

$$f(x|\alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}$$

Where $\Gamma(\alpha)$ is the *Gamma function*.

Normal (Gaussian) Distribution

The normal distribution is perhaps the most common probability distribution, occurring very often in nature.

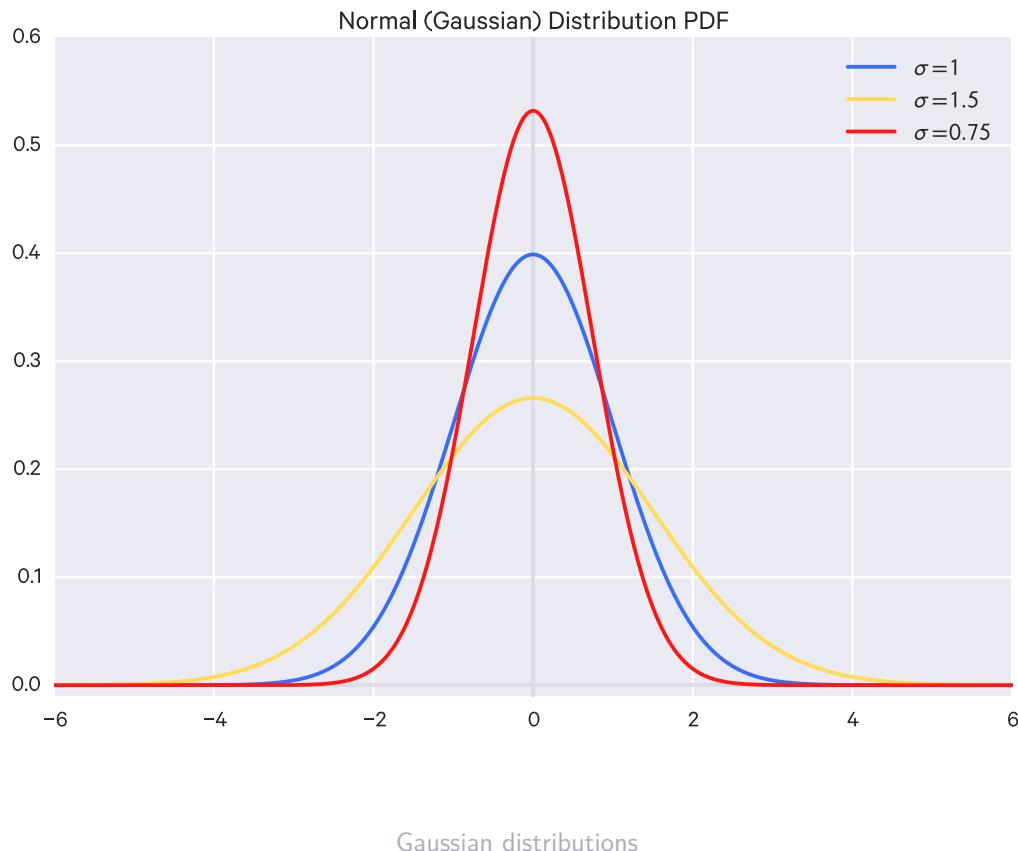
For a random variable x , the normal probability density function is¹:

¹ $\exp(x) = e^x$

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

The (univariate) Gaussian distribution is parameterized by μ and σ (for multivariate Gaussian distributions, see below).

The peak of the distribution is where $x = \mu$.



The height and width of the distribution varies according to σ . The lower σ is, the higher and thinner the distribution is; the higher σ is, the lower and wider the distribution is.

The *standard* normal distribution is just $N(0, 1)$.

The Gaussian distribution can be used to approximate other distributions, such as the binomial distribution when the number of experiments is large, or the Poisson distribution when the average arrival rate is high.

A normal random variable X is denoted:

$$X \sim N(\mu, \sigma)$$

Where the parameters are:

- μ = the mean

- σ = the standard deviation

The expected value is:

$$E[X|\mu, \sigma] = \mu$$

t Distribution

For small sample sizes ($n < 30$), the distribution of the sample mean deviates slightly from the normal distribution since the sample mean doesn't exactly match the population's mean. This distribution is the t-distribution.

This distribution is the t-distribution, which, for large enough sample sizes ($>= 30$), converges to the normal distribution, so it may also be used for large sample sizes too.

The t-distribution has thicker tails than the normal distribution, so observations are more likely to be within two standard deviations of its mean. This allows for more accurate estimations of the standard error for small sample sizes.

The t-distribution is always centered around zero and is described by one parameter: the **degrees of freedom**. The higher the degrees of freedom, the closer the t-distribution is to the standard normal distribution.

The confidence interval is computed slightly differently for a t distribution. Instead of the Z score we use a cutoff, t_{df} , determined by the degrees of freedom for the distribution.

For a single sample with n observations, the degrees of freedom is $df = n - 1$. For two samples, you can use a computer to calculate the degrees of freedom, or you can choose the smallest sample size minus 1.

The t-distribution's corresponding test is the t-test, sometimes called the "Student t-test", which is used to compare the means of two groups.

From the t-distribution we can calculate a t value:

$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}}$$

Then we can use this t value with the t distribution with the degrees of freedom for the sample and use that to compute a p-value.

Beta Distribution

For an event with two outcomes, the beta distribution is the probability distribution of the probability of the outcome being positive. The beta distribution's domain is $[0, 1]$ which makes it appropriate for this use.

That is, in a beta distribution both the y and the x axes represent probabilities. The x -axis is the possible probabilities for the events in question, and the y -axis is the probability that that possible probability is the true probability.

It is notated:

$$\text{Beta}(\alpha, \beta)$$

Where α is the number of positive examples and β is the number of negative examples.

Its PDF is:

$$f(x|\alpha, \beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$

Where $B(\alpha, \beta)$ is the *Beta function*.

The Beta distribution is a generalization of the uniform distribution:

$$\text{Uniform}() \sim \text{Beta}(1, 1)$$

The mean of a beta distribution is just $\frac{\alpha}{\alpha+\beta}$ which is pretty straightforward if you think about it.

If you need to estimate the probability of something happening, the beta distribution can be a good prior since it is quite easy to calculate its posterior distribution:

$$\text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}) = \text{Beta}(\alpha_{\text{likelihood}} + \alpha_{\text{prior}}, \beta_{\text{likelihood}} + \beta_{\text{prior}})$$

That is, you just use some plausible prior values for α and β such that you have a plausible mean, then just add your new positive and negative examples to update the beta distribution.

Weibull Distribution

The Weibull distribution is used for modeling reliability or “survival” data, e.g. for dealing with failure-rates.

It is defined as:

$$f_x(x) = \begin{cases} k \frac{x^{k-1}}{\lambda^k} e^{-(x/\lambda)^k} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

The k parameter is the *shape parameter* and the λ parameter is the *scale parameter* of the distribution.

If x is the “time-to-failure”, the Weibull distribution describes the failure rate over time. In this case, the parameter k influences how the failure rate changes over time: if $k < 1$, the failure rate decreases

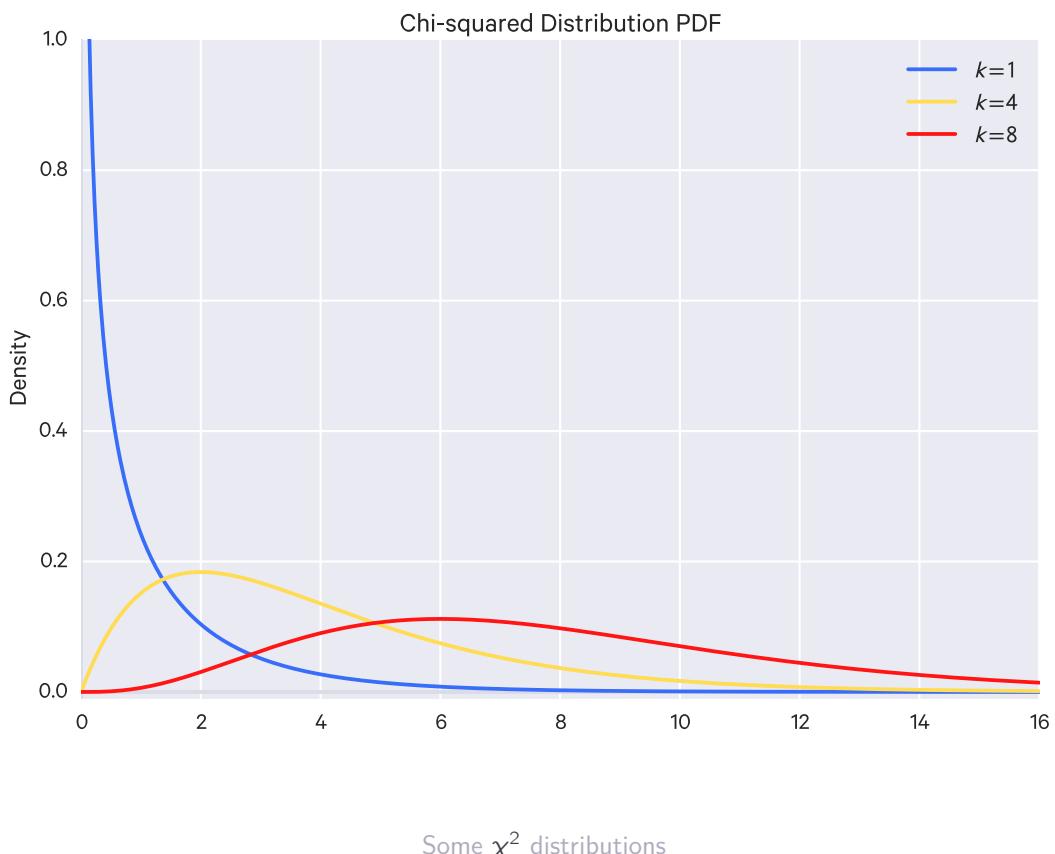
over time (for instance, defective products fail early and are weeded out), if $k = 1$, the failure rate is constant, and if $k > 1$, then the failure rate increases with time (e.g. parts degrade over time).

Chi-square (χ^2) distribution

The χ^2 distribution is closely related to the normal distribution and often used as a sampling distribution. The χ^2 distribution with f degrees of freedom, sometimes notated $\chi_{[f]}^2$, is the sum of the squares of f independent standard normal (e.g. $\mu = 0, \sigma = 1$) variates, i.e.:

$$Y = X_1^2 + X_2^2 + \cdots + X_f^2$$

This distribution has a mean f and a variance of $2f$ (this comes from the additive property of the mean and the variance). The skewness of the distribution follows the same additive property and is $\sqrt{\frac{8}{f}}$. So when f is small, the distribution skews to the right, and the skewness decreases as f increases. When f is very large the distribution approaches the standard normal distribution (by the central limit theorem).



4.13 Multiple random variables

In the real world you often work with multiple random variables simultaneously - that is, you are working in higher dimensions. You could describe a group of random variables as a *random vector*,

i.e. a random vector $X \in \mathbb{R}^d$, where d is the number of dimensions (the number of random variables) you are working in, i.e. $X = [X_1, \dots, X_d]$.

A distribution over multiple random variables is called a *joint distribution*.

For a joint distribution $P(a, b)$, the distribution of a subset of variables is called a *marginal distribution* (or just *marginal*) of the joint distribution, and is computed:

$$P(a) = \sum_b P(a, b)$$

That is, fix b to each of its possible outcomes and sum those probabilities.

Generally, you can compute the marginal like so:

$$P(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = \sum_{x_i} P(x_1, \dots, x_n)$$

So you take the variable you want to remove and sum over the probabilities with it fixed for each of its possible outcomes.

The distribution over multiple random variables is called a **joint distribution**. When we have multiple random variables, the distribution of some subset of those random variables is the **marginal distribution** for that subset.

The probability density function for a joint distribution just takes more arguments, i.e.:

$$P(a_1 \leq X_1 \leq b_1, a_2 \leq X_2 \leq b_2, \dots, a_n \leq X_n \leq b_n) = \int_{a_1}^{b_1} \int_{a_2}^{b_2} \cdots \int_{a_n}^{b_n} f(x_1, x_2, \dots, x_n) dx_1 dx_2 \dots dx_n$$

4.13.1 Conditional distributions

Conditional distributions are distributions in which the value of one or more other random variables are known.

For random variables X, Y , the conditional probability of $X = a$ given $Y = b$ is:

$$P(X = a | Y = b) = \frac{P(X = a, Y = b)}{P(Y = b)}$$

which is undefined if $P(Y = b) = 0$.

This can be expanded to multiple given random variables:

$$P(X = a | Y = b, Z = c) = \frac{P(X = a, Y = b, Z = c)}{P(Y = b, Z = c)}$$

The conditional distribution of X , conditioned on $Y = b$, is notated $P(X|Y = b)$.

More generally, we can describe the conditional distribution of X conditioned on Y on all its values as $P(X|Y)$.

For continuous random variables, the probability of the random variable being a given specific value is 0 (see the section on probability density functions), so here we have the denominator as 0, which won't do. However, it can be shown that the probability density function $f(y|x)$ underlying the distribution $P(Y|X)$ is given by:

$$f(y|x) = \frac{f(x,y)}{f(x)}$$

And thus:

$$P(a \leq Y \leq b|X = c) = \int_a^b f(y|c)dy = \int_a^b \frac{f(c,y)}{f(c)}dy$$

4.13.2 Multivariate Gaussian

A random vector X is (multivariate) Gaussian (or “normal”) if any linear combination is (univariate) Gaussian.

Note that “Gaussian” often implies “multivariate Gaussian”.

That is, the dot product of some vector a transpose with X , which is:

$$a^T X = \sum_{i=1}^n a_i X_i$$

is Gaussian for every $a \in \mathbb{R}^n$.

We say X is (multivariate) Gaussian distributed with mean μ (where $\mu \in \mathbb{R}^n$, that is, μ is a vector as well) and covariance matrix C , notated:

$$X \sim N(\mu, C)$$

which means X is Gaussian with $E[X_i] = \mu_i$ and $C_{ij} = \text{Cov}(X_i, X_j)$ and $C_{ii} = \text{Var}(X_i)$.

μ and C are the parameters of the distribution.

If X is a random vector, $X \in \mathbb{R}^n$, i.e. $[X_1, \dots, X_n]$, and a Gaussian, i.e. $X \sim N(\mu, C)$ where μ is a vector $\mu = [\mu_1, \dots, \mu_n]$, and if the covariance matrix C has the variances on its diagonal and 0 everywhere else, like below, then the components of X are independent and individually normally distributed, i.e. $X_i \sim N(\mu_i, \sigma_i^2)$.

Caveat: a random vector's individual components being Gaussian but *not* independent does not necessarily imply that the vector itself is Gaussian.

$$C = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & \cdots & 0 & \sigma_n^2 \end{pmatrix} = \text{diag}(\sigma_1^2, \dots, \sigma_n^2)$$

Intuitively this makes sense because if X_i and X_j are independent, then their covariance $\text{Cov}(X_i, X_j) = 0$. So all the i, j entries in C where $i \neq j$ are 0. This does not necessarily hold for non-Gaussians; this is a property particular to Gaussians.

Degenerate univariate Gaussian

A *degenerate* univariate Gaussian distribution is one where $X \equiv \mu$, that is: $X \sim N(\mu, 0)$.

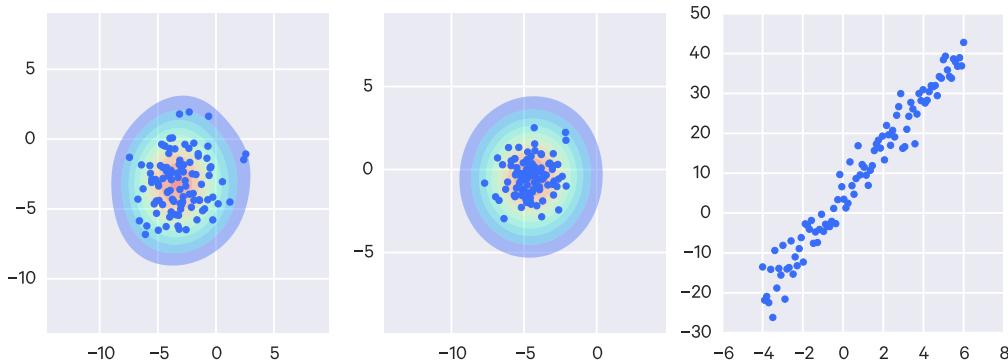
Degenerate multivariate Gaussian

A multivariate Gaussian can also be degenerate, which is when the determinant of its covariance matrix C is 0.

Examples of Gaussians (and non-examples)

These are some examples of what Gaussians can look like. Drawn over the first two are their *level sets* which demarcate where the density is constant (you can think of it like a topographical map).

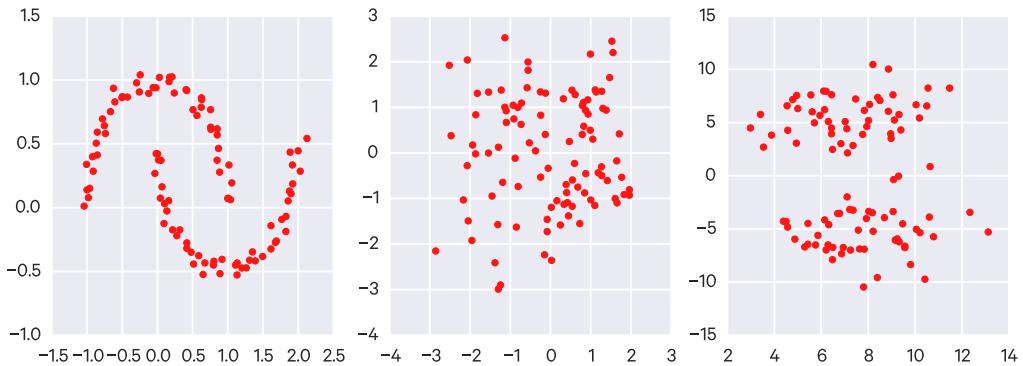
The last example is a degenerate Gaussian.



Some example Gaussians. The last example is a degenerate Gaussian

Probability density function

A multivariate Gaussian random variable $X \sim N(\mu, C)$ only has a density function if it is nondegenerate (i.e. $\det(C) \neq 0$).



Some examples that are not Gaussians

The PDF is:

$$\frac{1}{\sqrt{|2\pi C|}} \exp\left(-\frac{1}{2}(x - \mu)^T C^{-1}(x - \mu)\right)$$

Note that $|A| = \det(A)$, and the term $|2\pi C|$ can be also written as $(2\pi)^n \det(C)$.

Affine property

An *affine transformation* is just some function in the form $f(x) = Ax + b$.

Any affine transformation of a Gaussian random variable is itself a Gaussian. If $X \sim N(\mu, C)$, then $AX + b \sim N(A\mu + b, ACA^T)$.

Marginal distributions of a Gaussian

The marginal distributions of a Gaussian are also Gaussian.

More formally, if you have a Gaussian random vector $X \in \mathbb{R}^n$, $X \sim N(\mu, C)$ which you decompose into $X_a = [X_1, \dots, X_k]$, $X_b = [X_{k+1}, \dots, X_n]$, where $1 \leq k \leq n$, then $X_a \sim N(\mu_a, C_{aa})$, $X_b \sim N(\mu_b, C_{bb})$.

Conditional distributions of a Gaussian

The conditional distributions of a Gaussian are also Gaussian.

More formally, if you have a Gaussian random vector $X \in \mathbb{R}^n$, $X \sim N(\mu, C)$ which you decompose into $X_a = [X_1, \dots, X_k]$, $X_b = [X_{k+1}, \dots, X_n]$, where $1 \leq k \leq n$, then $(X_a | X_b = x_b) \sim N(m, D)$, where $m = \mu_a + C_{ab}C_{bb}^{-1}(x_b - \mu_b)$ and $D = C_{aa} - C_{ab}C_{bb}^{-1}C_{ba}$.

Sum of independent Gaussians

The sum of independent Gaussians is also Gaussian.

More formally, if you have Gaussian random vectors $X \in \mathbb{R}^n, X \sim N(\mu_x, C_x)$ and $Y \in \mathbb{R}^n, Y \sim N(\mu_y, C_y)$ which are independent, then $X + Y \sim N(\mu_x + \mu_y, C_x + C_y)$

4.14 Bayes' Theorem

4.14.1 Intuition

The probability of both a and b occurring is $P(a \cap b)$ (the probability of a or b occurring).

This is the same as the probability of a occurring given b and vice versa:

$$P(a \cap b) = P(a|b)P(b) = P(b|a)P(a)$$

This can be rearranged to form **Bayes' Theorem**:

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)}$$

Bayes' Theorem is useful for answering questions such as, "How likely is A given B?". For example, "How likely is my hypothesis true given the data I have?"

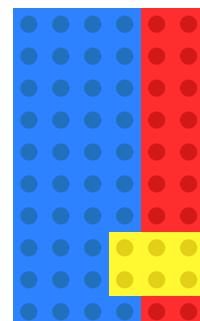
4.14.2 A Visual Explanation

This explanation is adapted from [Count Bayesie](#).

The accompanying figure depicts a 6x10 area (60 pegs total) of lego bricks representing a probability space with the following probabilities:

$$P(\text{blue}) = 40/60 = 2/3$$

$$P(\text{red}) = 20/60 = 1/3$$



Lego Probability Space

Red and blue alone describe the entire set of possible events.

Yellow pegs are *conditional* upon the red and blue bricks; that is, their probabilities are conditional upon what color brick is underneath it.

So the following probability properties of yellow should be straightforward:

$$P(\text{yellow}) = 6/10 = 1/10$$

$$P(\text{yellow}|\text{red}) = 4/20 = 1/5$$

$$P(\text{yellow}|\text{blue}) = 2/40 = 1/20$$

But say you want to figure out $P(\text{red}|\text{yellow})$. This is intuitive visually in this example. You'd reason that there are 6 yellow pegs total, 4 of which are on the red space, so there's 4/6 probability that we are in the red space for a given yellow peg.

This intuition is Bayes' Theorem, and can be written more formally as:

$$P(\text{red}|\text{yellow}) = \frac{P(\text{yellow}|\text{red})P(\text{red})}{P(\text{yellow})}$$

Step by step, what we did was:

$$\begin{aligned} n_{\text{yellow}} &= P(\text{yellow}) * n_{\text{total}} = 1/10 * 60 = 6 \\ n_{\text{red}} &= P(\text{red}) * n_{\text{total}} = 1/3 * 60 = 20 \\ n_{\text{yellow}|\text{red}} &= P(\text{yellow}|\text{red}) * n_{\text{red}} = 1/5 * 20 = 4 \\ P(\text{red}|\text{yellow}) &= \frac{n_{\text{yellow}|\text{red}}}{n_{\text{yellow}}} = 4/6 = 2/3 \end{aligned}$$

If you expand out the last equation, you'll find Bayes' Theorem:

$$\begin{aligned} P(\text{red}|\text{yellow}) &= \frac{n_{\text{yellow}|\text{red}}}{n_{\text{yellow}}} \\ &= \frac{P(\text{yellow}|\text{red}) * n_{\text{red}}}{P(\text{yellow}) * n_{\text{total}}} \\ &= \frac{P(\text{yellow}|\text{red}) * P(\text{red}) * n_{\text{total}}}{P(\text{yellow}) * n_{\text{total}}} \\ &= \frac{P(\text{yellow}|\text{red})P(\text{red})}{P(\text{yellow})} \end{aligned}$$

4.14.3 An Example Bayes' Problem

Consider the following problem:

1% of women at age forty who participate in routine screening have breast cancer. 80% of women with breast cancer will get positive mammographies. 9.6% of women without breast cancer will also get positive mammographies. A woman in this age group had a positive mammography in a routine screening. **What is the probability that she actually has breast cancer?**

Intuitively it's difficult to get the correct answer. Generally, only ~15% doctors can get it right (Casscells, Schoenberger, and Grayboys 1978; Eddy 1982; Gigerenzer and Hoffrage 1995; and many other studies.)

You can work through the problem like so:

- **1% of women at age forty have breast cancer.** To simplify the problem, assume there are 1000 women total, so 10/1000 have breast cancer.
- **80% of women w/ breast cancer will get positive mammographies.** So of the 10 women that have breast cancer, 8/1000 of them will get positive mammographies.
- **9.6% of women without breast cancer will also get positive mammographies.** We have 10/1000 women with breast cancer, which means there are 990 without breast cancer. Of those 990, 9.6% will also get positive mammographies, so ~95/1000 women are false positives.

We can rephrase the problem like so: **What is the probability that a woman in this age group has breast cancer, if she gets a positive mammography?**

In total, the number of positives we have are $95 + 8 = 103$. Then we can just use simple probability: there's an $8/103$ chance (7.8%) that she has breast cancer, and a $95/103$ chance (92.2%) that she's a false positive.

One way to interpret these results is that, in general, women of age forty have a 1% chance of having breast cancer. Getting a positive mammography does not indicate that you have breast cancer, it just "slides up" your probability of having it to 7.8%.

We could break up the group of 1000 women into:

- True positives: 8
- False positives: 95
- True negatives: $990 - 95 = 895$
- False negatives: $10 - 8 = 2$

Which totals to 1000, so everyone is accounted for.

4.14.4 Solving the problem with Bayes' Theorem

The original proportion of patients w/ breast cancer is the **prior probability**.

The probability of a **true positive** and the probability of a **false positive** are the **conditional probabilities**.

Collectively, this information is known as the **priors**. The priors are required to solve a Bayesian problem.

The final answer - the estimated probability that a patient has breast cancer given a positive mammography - is the revised probability, better known as the **posterior probability**.

If the two conditional probabilities are equal, the posterior probability equals the prior probability (i.e. if there's an equal chance of getting a false and a negative positive, then the test really tells you nothing).

4.14.5 Another Example

Your friend reads you a study which found that only 10% of happy people are rich. Your friend concludes that money can't buy happiness. How could you show them otherwise?

Rather than asking “What percent of happy people are rich?”, it is probably better to ask “What percent of rich people are happy?” to determine if money buys happiness.

With the statistic from the study, statistics about the overall rate of happy people (say 40% of people are happy) and rich people (say 5% of people are rich), and Bayes’ Theorem, you can calculate this value:

$$10\% \times \frac{40\%}{5\%} = 80\%$$

So it seems like a lot of rich people are happy.

4.14.6 Naive Bayes

Bayes’ rule:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

Say a is a class and b is some evidence.

We’ll notate the class as c and the evidence as e . We are interested in: what’s the probability of a class c given some evidence e ? We can write this question out as Bayes’ rule:

$$P(c|e) = \frac{P(e|c)P(c)}{P(e)}$$

Our evidence may actually be multiple pieces of evidence: e_1, \dots, e_n . So instead we can re-write the equation as:

$$\frac{P(e_1, \dots, e_n|c)P(c)}{P(e_1, \dots, e_n)}$$

If we can assume that each piece of evidence is independent given the class c , then we can further write this as:

$$\frac{[\prod_i^n P(e_i|c)]P(c)}{P(e_1, \dots, e_n)}$$

Example

In practice: say I have two coins. One is a fair coin ($P(\text{heads}) = 0.5$) and one is a trick coin ($P(\text{heads}) = 0.8$). I pick one of the coins at random and flip it twice, getting heads and then tails. Which coin did I pick?

The head and tail outcomes are our evidence. So we can take the product of the probabilities of these outcomes given a particular class.

The probability of picking either coin was uniform, i.e. there was a 50% chance of picking either. So we can ignore that probability.

For a fair coin, the probability of getting heads and then tails is $P(H|\text{fair}) \times P(T|\text{fair}) = 0.5 \times 0.5 = 0.25$. For the trick coin, the probability is $P(H|\text{trick}) \times P(T|\text{trick}) = 0.8 \times 0.2 = 0.16$. So it's more likely that I picked the fair coin.

If we flip again and get a heads, things change a bit:

For a fair coin: $P(H|\text{fair}) \times P(T|\text{fair}) \times P(H|\text{fair}) = 0.5 \times 0.5 \times 0.5 = 0.125$. For the trick coin: $P(H|\text{trick}) \times P(T|\text{trick}) \times P(H|\text{trick}) = 0.8 \times 0.2 \times 0.8 = 0.128$. So now it's slightly more likely that I picked the trick coin.

4.15 Entropy

Broadly, entropy is the measure of disorder in a system.

In the case of probability, it is the measure of uncertainty that is associated with the distribution of a random variable.

If there are a few outcomes which are fairly certain, the system has low entropy. A point-mass distribution has the lowest entropy. We know exactly what value we'll get from it.

If there are many outcomes which are equiprobable, the system has high entropy. A uniform distribution has the highest entropy. We don't really have any idea of what value we'll draw from it.

The entropy of a random variable X is denoted $H(X)$, must be $H(X) \geq 0$ and is calculated:

$$\begin{aligned} H(X) &= -E[\lg(P(X))] \\ &= -\sum_x P(x) \lg(P(x)) \quad (\text{discrete}) \\ &= -\int_{-\infty}^{\infty} P(x) \lg(P(x)) dx \quad (\text{continuous}) \end{aligned}$$

Where $\lg(x) = \log_2(x)$.

This does not say anything about the value of the random variable, only the spread of its distribution.

For example: what is the entropy of a roll of a six-sided die?

$$1\left(\frac{1}{6} \lg\left(\frac{1}{6}\right) + \frac{1}{6} \lg\left(\frac{1}{6}\right)\right) = 2.58$$

The *Maximum Entropy Principle* says that, all else being equal, we should prefer distributions that maximize the entropy. That is, you should be conservative in your confidence about how much you know - if you don't have any good reason for something to be more likely than something else, err on the side of them being equiprobable.

4.16 The log trick

When working with many independent probabilities, which is often the case in machine learning, you have to multiply many probabilities which can result in underflow. So it's often easier to work with the logarithm of probability functions, which is fine because when optimizing, the max (or min) will be at the same location in the logarithm form (though their actual values will be different). Using logarithms will allow us to sum terms instead of multiplying them.

4.17 References

- <https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers>
- <http://www.countbayesie.com/blog/2015/4/4/parameter-estimation-the-pdf-cdf-and-quantile-function>
- <http://stats.stackexchange.com/questions/47771/what-is-the-intuition-behind-beta-distribution/47782#47782>
- <http://work.thaslwanter.at/Stats/html/statsDistributions.html>
- Probability Theory Review for Machine Learning, Samuel Ieong. November 6, 2006.
- MIT 6.034 (Fall 2010): Artificial Intelligence, taught by Patrick H. Winston.
- *Principles of Statistics*, M.G. Bulmer. 1979.
- OpenIntro Statistics, Second Edition. David M Diez, Christopher D Barr, Mine Çetinkaya-Rundel.
- <http://deeplearning4j.org/eigenvector>
- <http://www.yudkowsky.net/rational/bayes>
- <https://www.countbayesie.com/blog/2015/4/23/why-so-square-jensens-inequality-and-moments-of-a-random-variable>
- <http://www.quora.com/Probability/How-do-you-explain-Bayes-Theorem-in-simple-words>
- <http://www.countbayesie.com/blog/2015/2/18/bayes-theorem-with-lego>

5

Statistics

Broadly, statistics is concerned with collecting and analyzing data. It seeks to describe rigorous methods for collecting data (samples), for describing the data, and for inferring conclusions from the data. There are processes out there in the world that generate observable data, but these processes are often black boxes and we want to gain some insight into how they work.

We can crudely cleave statistics into two main practices: **descriptive statistics**, which provides tools for *describing* data, and **inferential statistics**, which provides tools for learning (inferring or estimating) from data.

This section is focused on *frequentist* (or classical) statistics, which is distinguished from *Bayesian statistics* (covered in another chapter).

5.0.1 Notation

- Regular letters, e.g. X, Y , typically denote **observed** (known) variables
- Greek letters, e.g. μ, σ , typically denote **unknown** variables which we are trying to estimate
- Hats over letters, e.g. $\hat{\theta}$, denote estimators (an estimator is a rule for calculating an estimate given some observed data), e.g. an estimated value for a parameter.

5.1 Descriptive Statistics

Descriptive statistics involves computing values which summarize a set of data. This typically includes statistics like the mean, standard deviation, median, min, max, etc, which are called **summary statistics**.

5.1.1 Scales of Measurement

In statistics, numbers and variables are categorized in certain ways. These are *scales of measurement*.

- **Nominal or Categorical:** These are qualitative variables; numbers here are arbitrarily assigned to represent categories of qualities. Nominal variables can only be counted; they have no order or intervals.
 - Example: Gender, marital status
- **Ordinal:** Ordinal variables have a concept of order, so they can be arranged into some sequence accordingly and meaningfully ranked. But they are without any measure of magnitude between items in that sequence. So some object A may come after some object B but there is no measurement of interval between the two (we can't, for instance, say that A is 10 more than B).
 - Example: Education level (some high school, high school, college, etc)
- **Interval:** Interval variables are like ordinal variables but do have a measure of interval between items. But they do not have an absolute zero point, so we can't compare values as ratios (we can't, for instance, say A is twice of B).
 - Example: Dates (we can say how many days there are between two dates, but, for example, we can't say one date is twice of another)
- **Ratio:** Ratio variables are like interval variables but have a fixed and meaningful zero point, so they can be compared as ratios.
 - Example: Age, length

5.1.2 Averages

The average of a set of data can be described as its **central tendency**; which gives some sense of a typical or common value for a variable. There are three types:

Arithmetic mean

Often just called the “mean” and notated μ (mu).

For a dataset $\{x_1, \dots, x_n\}$, the arithmetic mean is:

$$\frac{\sum_{i=1}^n x_i}{n}$$

The mean can be sensitive to extreme values (outliers), which is one reason the median is sometimes used instead. Which is to say, the median is a more **robust** statistic (meaning that it is less sensitive to outliers).

Note that there are other types of means, but the arithmetic mean is by far the most common.

Median

The central value in the dataset, e.g.

1 1 2 3 4

$$\text{median} = 2$$

If there are even number of values, you just take the value between the two central values:

1 1 2 3 4 4

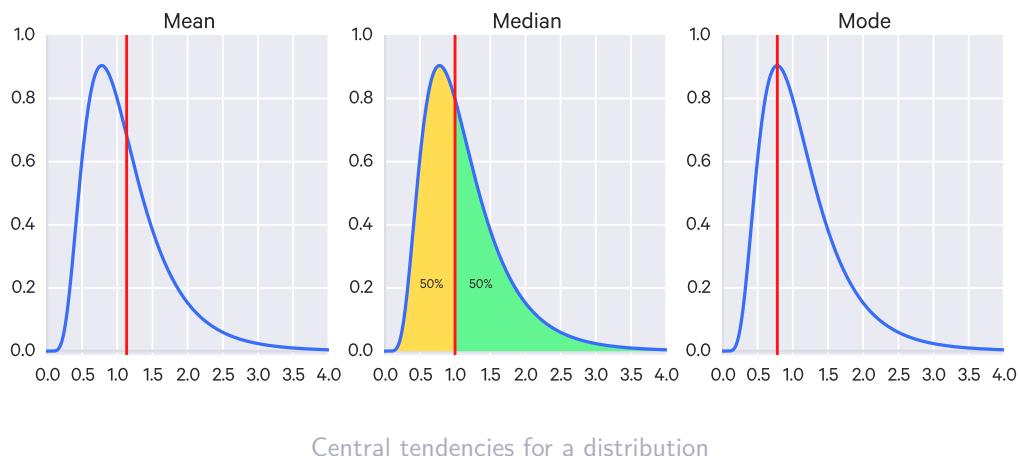
$$\text{median} = \frac{2+3}{2} = 2.5$$

Mode

The most frequently occurring value in the dataset, e.g.

1 2 3 3 2 3 4 3

$$\text{mode} = 3$$



5.1.3 Population vs Sample

With statistics we take a *sample* of a broader *population* or already have data which is a sample from a population. We use this limited sample in order to learn things about the whole population.

The mean of the population is denoted μ and consists of N items, whereas the mean of the sample (i.e. the *sample mean*, sometimes called the *empirical mean*) is notated \bar{x} or $\hat{\mu}$ and consists of n items.

The sample mean is:

$$\hat{\mu} = \frac{1}{n} \sum_i x^{(i)}$$

The sample variance is:

$$\hat{\sigma}^2 = \frac{1}{n-1} \sum_i (x^{(i)} - \hat{\mu})^2$$

The sample covariance matrix is:

$$\hat{\Sigma} = \frac{1}{n-1} \sum_i (x^{(i)} - \hat{\mu})(x^{(t)} - \hat{\mu})^T$$

These estimators are unbiased, i.e.:

$$\begin{aligned} E[\hat{\mu}] &= \mu \\ E[\hat{\sigma}^2] &= \sigma^2 \\ E[\hat{\Sigma}] &= \Sigma \end{aligned}$$

5.1.4 Independent and Identically Distributed

Often in statistics we assume that a sample is **independent and identically distributed** (iid); that is; that the data points are independent from one another (the outcome of one has no influence over the outcome of any of the others) and that they share the same distribution.

We say that X_1, \dots, X_n are iid if they are independent and drawn from the same distribution, that is $P(X_1) = \dots = P(X_n)$. This can also be stated:

$$P(X_1, \dots, X_n) = \prod_i P(X_i)$$

In this case, they all share the same mean (expected value) and variance.

This assumption makes computing statistics for the sample much easier.

For instance, if a sample was not identically distributed, each datapoint might come from a different distribution, in which case there are different means and variances for each datapoint which must be computed from each of those datapoints alone. They can't really be treated as a group since the datapoints aren't quite equivalent to each other (in a statistical sense).

Or, if the sample was not independent, then we lose all the conveniences that come with independence. The IID assumption doesn't always hold (i.e. it may be violated), of course, so there are other ways of approaching such situations while minimizing complexity, such as Hidden Markov Models.

5.1.5 The Law of Large Numbers (LLN)

Let X_1, \dots, X_n be iid with mean μ .

The **law of large numbers** essentially states that as a sample size approaches infinity, its mean will approach the population ("true") mean:

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n X_i = \mu$$

5.1.6 Regression to the mean

- $P(Y < x | X = x)$ gets bigger as x approaches very large values. That is, given a very large X (an extreme), the chance that Y 's value is as large as or larger than X is unlikely.
- $P(Y > x | X = x)$ gets bigger as x approaches very small values. That is, given a very small X (an extreme), the chance that Y 's value is as small as or smaller than X is unlikely.

5.1.7 Central Limit Theorem (CLT)

Say you have a set of data. Even if the distribution of that data is not normal, you can divide the data into groups (samples) and then average the values of those groups. Those averages will approach the form of a normal curve as you increase the size of those groups (i.e. increase the sample size).

Let X_1, \dots, X_n be iid with mean μ and variance σ^2 .

Then the central limit theorem can be formalized as:

$$\sqrt{\frac{n}{\sigma^2}} \left(\left(\frac{1}{n} \sum_{i=1}^n X_i \right) - \mu \right) \xrightarrow{D} N(0, 1)$$

That is, the left side *converges in distribution* to a normal distribution with mean 0 and variance 1 as n increases.

5.1.8 Dispersion (Variance and Standard Deviation)

Dispersion is the "spread" of a distribution - how spread out its values are.

The main measures of dispersion are the variance and the standard deviation.

Standard Deviation

Standard deviation is represented by σ (sigma) and describes the variation from the mean (μ , i.e. the expected value), calculated:

$$\sigma = \sqrt{E[(X - \mu)^2]} = \sqrt{E[X^2] - (E[X])^2}$$

Variance

The square of the standard deviation, that is, $E[(X - \mu)^2]$, is the **variance** of X , usually notated σ^2 . It can also be written:

$$\text{Var}(X) = \sigma^2 = E(X^2) - E(X)^2 = \frac{\sum_{i=1}^N (x_i - \mu)^2}{N}$$

For a population of size N of datapoints x .

That is, variance is the difference between the square of the inputs and the square of the expected value.

Coefficient of variation (CV)

Variance depends on the units of measurement, but this can be controlled for by computing the **coefficient of variation**:

$$CV = \frac{\sigma}{\bar{X}} \times 100$$

This allows us to compare variability across variables measured in different units.

Variance of a linear combination of random variables

The variance of a linear combination of (independent) random variables, e.g. $aX + bY$, can be computed:

$$\text{Var}(\beta_1 X_1 + \cdots + \beta_n X_n) = \beta_1^2 \text{Var}(X_1) + \cdots + \beta_n^2 \text{Var}(X_n)$$

Range

The **range** can also be used to get a sense of dispersion. The range is the difference between the highest and lowest values, but very sensitive to outliers.

As an alternative to the range, you can look at the **interquartile range**, which is the range of the middle 50% of the values (that is, the difference of the 75th and 25th percentile values). This is less sensitive to outliers.

Z score

A **Z score** is just the number of standard deviations a value is from the mean. It is defined:

$$Z = \frac{x - \mu}{\sigma}$$

The Empirical Rule

The **empirical rule** describes that, for a normal distribution, there is:

- a 68% chance that a value falls within one standard deviation
- a 95% chance that something falls within two standard deviations
- a 99.7% chance that something falls within three standard deviations

Pooled standard deviation estimates

If you have reason to expect that the standard deviations of two populations are practically identical, you can use the **pooled standard deviation** of the two groups to obtain a more accurate estimate of the standard deviation and standard error:

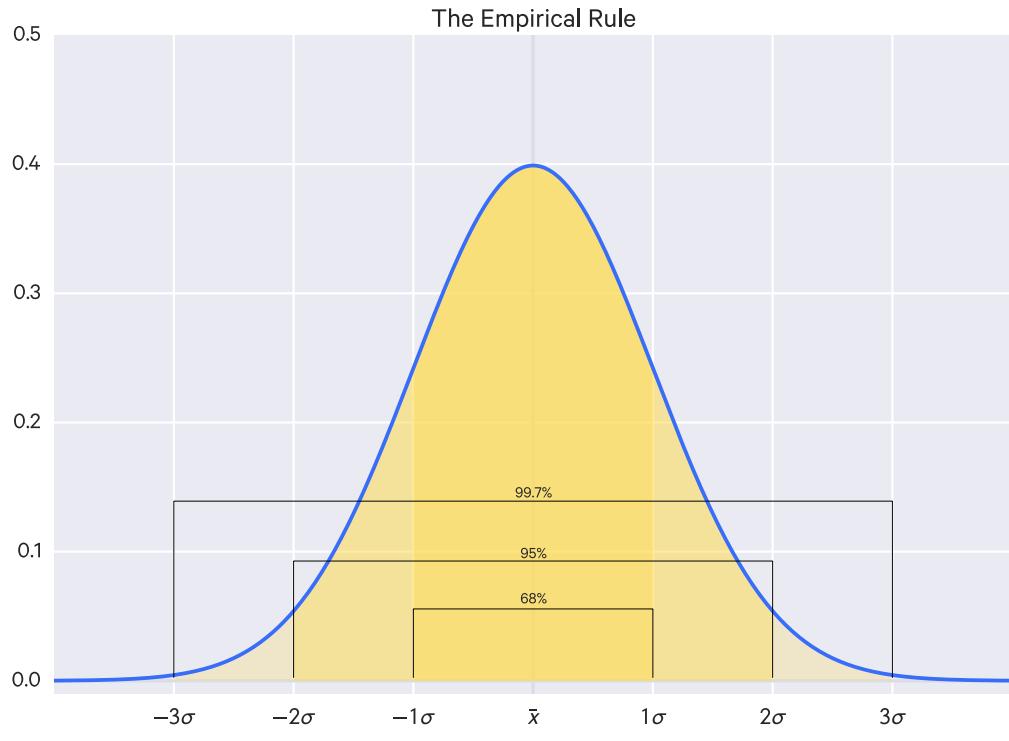
$$s_{\text{pooled}}^2 = \frac{s_1^2(n_1 - 1) + s_2^2(n_2 - 1)}{n_1 + n_2 - 2}$$

Where n_1, n_2, s_1, s_2 are the sample sizes and standard deviations of the sample groups. We must update the degrees of freedom as well, $\text{df} = n_1 + n_2 - 2$, which we can use for a new t-distribution.

5.1.9 Moments

The k th *moment*, $m_k(X)$, where $k \in 1, 2, 3, \dots$ (i.e. it is a positive integer), is $E[X^k]$. So the first moment is just the mean, $E[X]$.

The k th *central moment* is $E[(X - E[X])^k]$. So the second central moment is just the variance, $E[(X - E[X])^2]$.



The Empirical Rule

The third moment is the skewness, and the fourth moment is the kurtosis; they all share the same form (with different normalization terms):

$$\text{skewness} = E[(X - \mu)^3] \frac{1}{\sigma^3}$$

$$\text{kurtosis} = E[(X - \mu)^4] \frac{1}{\sigma^4}$$

Moments have different units, e.g. the first moment might be in meters (m), the second moment would be m^2 , and so on, so it is typical to standardize moments by taking their k -th root, e.g. $\sqrt{m^2}$

5.1.10 Covariance

The **covariance** describes the variance between two random variables.

For random variables x and y , the covariance is¹:

$$\text{Cov}(x, y) = E[(x - E(x))(y - E(y))] = \frac{1}{n} \sum (x_i - \bar{x})(y_i - \bar{y})$$

There must be the same number of values n for each.

This is simplified to:

¹Remember, $E(x)$ denotes the expected value of random variable x

$$\text{Cov}(x, y) = E[xy] - E[y]E[x] \approx \bar{xy} - \bar{y}\bar{x}$$

A positive covariance means that as x goes up, y goes up. A negative covariance means that as x goes up, y goes down.

Note that variance is just the covariance of a random variable with itself:

$$\text{Var}(X) = E(XX) - E(X)E(X) = \text{Cov}(X, X)$$

5.1.11 Correlation

Correlation gives us a measure of relatedness between two variables. Alone it does not imply causation, but it can help guide more formal inquiries (e.g. experiments) into causal relationships.

A good way to visually intuit correlation is through scatterplots.

We can measure correlation with *correlation coefficients*. These measure the strength and sign of a relationship (but not the slope, *linear regression*, detailed later, does that).

Some of the more common correlation coefficients include:

- Pearson product-moment (used where both variables are on an interval or ratio scale)
- Spearman rank-order (where both variables are on ordinal scales)
- Phi (where both variables are on nominal/categorical/dichotomous/binary scales)
- Point biserial (where one variable is on a nominal/categorical/dichotomous/binary scale and the other is on an interval or ratio scale)

The Pearson and Spearman coefficients are the most commonly used ones, but sometimes the latter two are used in special cases (e.g. with categorical data).

Pearson product-moment correlation coefficient (Pearson's correlation)

$$r = \frac{\sum_{i=1}^n \left(\frac{x_i - \bar{x}}{s_x} \right) \left(\frac{y_i - \bar{y}}{s_y} \right)}{n - 1}$$

Note: this is sometimes denoted as a capital R

You may recognize this as:

$$\frac{\text{Cov}(X, Y)}{S_X S_Y}$$

Here we convert our values to **standard scores**, i.e. $\frac{x_i - \bar{x}}{s_x}$. This standardizes the values such that their mean is 0 and their variance is 1 (and so they are unitless)

For a population, r is notated ρ (rho).

This value can range from $[-1, 1]$, where 1 and -1 mean complete correlation and 0 means no correlation.

To test the statistical significance of the Pearson correlation coefficient, you can use the t statistic.

For instance, if you believe there is a relationship between two variables, you set your null hypothesis as $\rho = 0$ and then with your estimate of r , calculate the t statistic:

$$t = \frac{r}{\sqrt{\frac{1-r^2}{n-2}}}$$

Then look up the value in a t table.

The Pearson correlation coefficient tells you the strength and direction of a relationship, but it doesn't tell you how much variance of one variable is explained by the other.

For that, you can use the *coefficient of determination* which is just r^2 . So for instance, if you have $r = 0.9$, then $r^2 = 0.81$ which means 81% of the variation of one variable is explained by the other.

Note that Pearson's correlation only accurately measures linear relationships; so even if you have a Pearson correlation near 0, it is still possible that there may be a strong nonlinear relationship. It's worthwhile to look at a scatter plot to verify.

It is also not robust in the presence of outliers.

Spearman rank-order correlation coefficient (Spearman's rank correlation)

Here you compute ranks (i.e. the indices in the sorted sample) rather than standard scores.

For example, for the dataset $[1, 10, 100]$, the rank of the value 10 is 2 because it is second in the sorted list.

Then you can compute the Spearman correlation:

$$r_s = 1 - \frac{(6 \sum d^2)}{n(n^2 - 1)}$$

Where d is the difference in ranks for each datapoint.

Generally, you can interpret r_s in the following ways:

- $0.9 \leq r_s \leq 1$ - very strong correlation
- $0.7 \leq r_s \leq 0.9$ - strong correlation
- $0.5 \leq r_s \leq 0.7$ - moderate correlation

You can test its statistical significance using a z test, where the null hypothesis is that $r_s = 0$.

$$z = r_s \sqrt{n - 1}$$

Spearman's correlation is more robust to outliers and skewed distributions.

Point-Biserial correlation coefficient

This correlation coefficient is useful when comparing a categorical (binary) variable with an interval or ratio scale variable:

$$r_{pbi} = \frac{M_p - M_q}{S_t} \sqrt{pq}$$

Where M_p is the mean for the datapoints categorized as 1 and M_q is the mean for the datapoints categorized as 0. S_t is the standard deviation for the interval/ratio variable, p is the proportion of datapoints categorized as 1, and q is the proportion of datapoints categorized as 0.

Phi correlation coefficient

This allows you to measure the correlation between two categorical (binary) variables.

It is calculated like so:

$$\begin{aligned} A &= f(0, 1) \\ B &= f(1, 1) \\ C &= f(0, 0) \\ D &= f(1, 0) \\ r_\phi &= \frac{AD - BC}{\sqrt{(A+B)(C+D)(A+C)(B+D)}} \end{aligned}$$

Where $f(a, b)$ is the frequency of label a and label b occurring together in the data.

5.1.12 Degrees of Freedom

Degrees of freedom describes the number of variables that are “free” in what value they can take. Often a given variable must be a particular value because of the values the other variables take on and some constraint(s).

For example: say we have four unknown quantities x_1, x_2, x_3, x_4 . We know that their mean is 5. In this case we have three degrees of freedom - this is because three of the variables are free to take arbitrary values, but once those three are set, the fourth value *must* be equal to $x_4 = 20 - x_1 - x_2 - x_3$ in order for the mean to be 5 (that is, in order to satisfy the constraint). So for instance, if $x_1 = 2, x_2 = 4, x_3 = 6$, then x_4 *must* equal 8. It is not “free” to take on any other value.

5.1.13 Time Series Analysis

Often data has a temporal component; e.g. you are looking for patterns over time.

Generally, time series data may have the following parts: a **trend**, which is some function reflecting persistent changes, **seasonality**; that is, periodic variation, and of course there is going to be some noise - random variation - as well.

Moving averages

To extract a trend from a series, you can use regression, but sometimes you will be better off with some kind of **moving average**. This divides the series into overlapping regions, **windows**, of some size, and takes the averages of each window. The **rolling mean** just takes the mean of each window. There is also the **exponentially-weighted moving average** (EWMA) which gives a weighted average, such that more recent values have the highest weight, and values before that have weights which drop off exponentially. The EWMA takes an additional **span** parameter which determines how fast the weights drop off.

Serial correlation (autocorrelation)

In time series data you may expect to see patterns. For example, if a value is low, it may stay low for a bit, if it's high, it may stay high for a bit. These types of patterns are **serial correlations**, also called **autocorrelation** (so-called because it is correlated a dataset with itself, in some sense), because the values correlate in their sequence.

You can compute serial correlation by shifting the time series by some interval, called a **lag**, and then compute the correlation of the shifted series with the original, unshifted series.

5.1.14 Survival Analysis

Survival analysis describes how long something lasts. It can refer to the survival of, for instance, a person - in the context of disease, a 5-year survival rate is the probability of surviving 5 years after diagnosis, for example - or a mechanical component, and so on. More broadly it can be seen as looking at how long something lasts until something happens - for instance, how long until someone gets married.

A **survival curve** is a function $S(t)$ which computes the probability of surviving longer than duration t . Such a duration is called a *lifetime*.

The survival curve ends up just being the complement of the CDF:

$$S(t) = 1 - \text{CDF}(t)$$

Looking at it this way, the CDF is the probability of a lifetime *less than or equal* to t .

Hazard function

A **hazard function** tells you the fraction of cases that continue until t and then end at t . It can be computed from the survival curve:

$$\lambda(t) = \frac{S(t) - S(t+1)}{S(t)}$$

Hazard functions are also used for estimating survival curves.

Estimating survival curves: Kaplan-Meier estimation

Often we do not have the CDF of lifetimes so we can't easily compute the survival curve. We often have non-survival cases alongside have survival cases, where we don't yet know what their final lifetime will be. Often, as is the case in the medical context, we don't want to wait to learn what these unknown lifetimes will be. So we need to estimate the survival curve with the data we do have.

The Kaplan-Meier estimation allows us to do this. We can use the data we have to estimate the hazard function, and then convert that into a survival curve.

We can convert a hazard function into an estimate of the survival curve, where each point at time t is computed by taking the product of complementary hazard functions through that time t , like so:

$$\prod_t (1 - \lambda(t))$$

5.2 Inferential Statistics

Statistical inference is the practice of using statistics to *infer* some conclusion about a population based on only a sample of that population. This can be the population's distribution – we want to infer from the sample data what the “true” distribution (the population distribution) is and the unknown parameters that define it.

Generally, data is generated by some *process*; this data-generating process is also *noisy*; that is, there is a relatively small degree of imprecision or fluctuation in values due to randomness. In inferential statistics, we try to uncover the particular function that describes this process as closely as possible. We do so by choosing a *model* (e.g. if we believe it can be modeled linearly, we might choose linear regression, otherwise we might choose a different kind of model such as a probability distribution; modeling is covered in greater detail in the machine learning part). Once we have chosen the model, then we need to determine the *parameters* (linear coefficients, for example, or mean and variance for a probability distribution) for that model.

Broadly, the two paradigms of inference are **frequentist**, which relies on long-run repetitions of an event, that is, it is *empirical* (and could be termed the “conventional” or “traditional” framework, though there’s a lot of focus on Bayesian inference now) and **Bayesian**, which is about generating a

hypothesis distribution (the prior) and updating it as more evidence is acquired. Bayesian inference is valuable because there are many events which we cannot repeat, but we still want to learn something about.

The frequentist believes these unknown parameters have precise “true” values which can be (approximately) uncovered. In frequentist statistics, we can estimate these exact values. When we estimate a single value for an unknown, that estimation is called a *point estimate*. This is in contrast to describing a value estimate as a probability distribution, which is the Bayesian method. The Bayesian believes that we cannot express these parameters as single values and we should rather describe them as a distributions of possible values to be explicit about their uncertainty.

Here we focus on frequentist inference; Bayesian inference is covered in a later chapter.

In frequentist statistics, the factor of noise means that we may see relationships (and thus come up with non-zero parameters) where they don’t exist, just because of the random noise. This is what p-values are meant to compensate for - if the relationship truly did not exist, what’s the probability, given the data, that we’d see the non-zero parameter estimate that we computed? Generally if this probability is less than 0.05 (i.e. $p < 0.05$) then we accept the result.

Often with statistical inference you are trying to quantify some difference between groups (which can be framed as measuring an *effect size*) or testing if some data supports or refutes some hypothesis, and then trying to determine whether or not this difference or effect can be attributed to chance (this is covered in the section on experimental statistics).

A word of caution: many statistical tools work only under certain conditions, e.g. assumptions of independence, or for a particular distribution, or a large enough sample size, or lack of skew, and so on - so before applying statistical methods and drawing conclusions, make sure the tools are appropriate for the data. And of course you must always be cautious of potential biases involved in the data collection process.

5.2.1 Error

Dealing with error is a big part of statistics and some error is unavoidable (noise is natural).

There are three kinds of error:

- Systemic error (systemic flaws in the data collection, e.g. sampling bias)
- Measurement error (due to imprecise instruments, for instance)
- Random error (natural noise, due to chance, uncontrollable, but in theory its effect is minimized if many measurements are taken)

We never know the true value of something, only what we observe by imprecise means, so we always must grapple with error.

5.2.2 Estimates and estimators

We can think of the population as representing the underlying data generating process and consider these parameters as functions of the population. To **estimate** these parameters from the sample data,

we use **estimators**, which are functions of the sample data that return an estimate for some unknown value. Essentially, any statistic is an estimator. For instance, we may estimate the population mean by using the sample mean as our estimator. Or we may estimate the population variance as the sample variance. And so on.

Estimators may be **biased** for small sample sizes; that is, it tends to have more error for small sample sizes. There are **unbiased** estimators as well, which have an expected mean error (against the population parameter) of 0. For example, an unbiased estimator for population variance σ^2 is:

$$\frac{1}{n-1} \sum (x_i - \bar{x})^2$$

For an estimate, we can measure its **standard error (SE)**, which describes how much we expect the estimate to be off by, on average. Much of statistical inference is concerned with measuring the quality of these estimates.

5.2.3 Point Estimation

Given an unknown population parameter, we may want to estimate a single value for it - this estimate is called a **point estimate**. Ideally, the estimate is as close to the true value as possible.

The estimation formula (the function which yields an estimate) is called an **estimator** and is a random variable (so there is some underlying distribution). A particular value of the estimator is the **estimate**.

A simple example: we have a series of trials with some number of successes. We want an estimate for the probability of success of the event we looked at. Here an obvious estimate is be the number of successes over the total number of trials, so our estimator would be $\frac{X}{N}$ and - say we had 40 successes out of 100 trials - our estimate would be 0.4.

We consider a “good” estimator one whose distribution is concentrated as closely as possible around the parameter’s true value (that is, it has a small variance). Generally this becomes the case as more data is collected.

We can take multiple samples (of a fixed size) from a population and compute a point estimate (e.g. for the mean) from each. Then we can consider the distribution of these point estimates - this distribution is called a **sampling distribution**. The standard deviation of the sampling distribution describes the typical error of a point estimate, so this standard deviation is known as the **standard error (SE)** of the estimate.

Alternatively, if you have only one sample, the standard error of the sample mean \bar{x} can be computed (where n is the size of the sample):

$$SE_{\bar{x}} = \sigma_{\bar{x}} = \frac{\sigma_x}{\sqrt{n}}$$

This however requires the population standard deviation, σ_x , which probably isn’t known - but we can also use a point estimate for that as well; that is, you can just use s , the sample standard deviation,

instead (provided that the sample size is at least 30, as a rule of thumb, and the population distribution is not strongly skewed).

Also remember that the distribution of sample means approximates a normal distribution, with better approximation as sample size increases, as described by the central limit theorem. Some other point estimates' sampling distribution also approximate a normal distribution. Such point estimates are called **normal point estimates**.

There are other such computations for the standard error of other estimates as well.

We say a point estimate is **unbiased** if the sampling distribution of the estimate is centered at the parameter it estimates.

5.2.4 Nuisance Parameters

Nuisance parameters are values we are not directly interested in, but still need to be dealt with in order to get at what we *are* interested in.

5.2.5 Confidence Intervals

Rather than provide a single value estimate of a population parameter, that is, a point estimate, it can be better to provide a range of values for the estimate instead. This range of values is a **confidence interval**. The confidence interval is the range of values where an estimate is likely to fall with some percent probability.

Confidence intervals are expressed in percentages, e.g. the “95% confidence interval”, which describes the plausibility that the parameter is in that interval. It *does not* imply a probability (that is, it does not mean that the true parameter has a 95% chance of being in that interval), however. Rather, the 95% confidence interval is the range of values in which, over repeated experimentation, in 95% of the experiments, that confidence interval will contain the true value. You would say “We are 95% confident the population parameter is in this interval”.

Confidence intervals are a tool for frequentist statistics, and in frequentist statistics, unknown parameters are considered fixed (we don't express them in terms of probability as we do in Bayesian statistics). So we do not associate a probability with the parameter. Rather, the *confidence interval itself* is the random variable, not the parameter. To put it another way, we are saying that 95% of the intervals we would generate from repeated experimentation would contain the real parameter - but we aren't saying anything about the parameter's value changing, just that the intervals will vary across experiments.

The mathematical definition of the 95% confidence interval is (where θ is the unknown parameter):

$$P(a(Y) < \theta < b(Y) | \theta) = 0.95$$

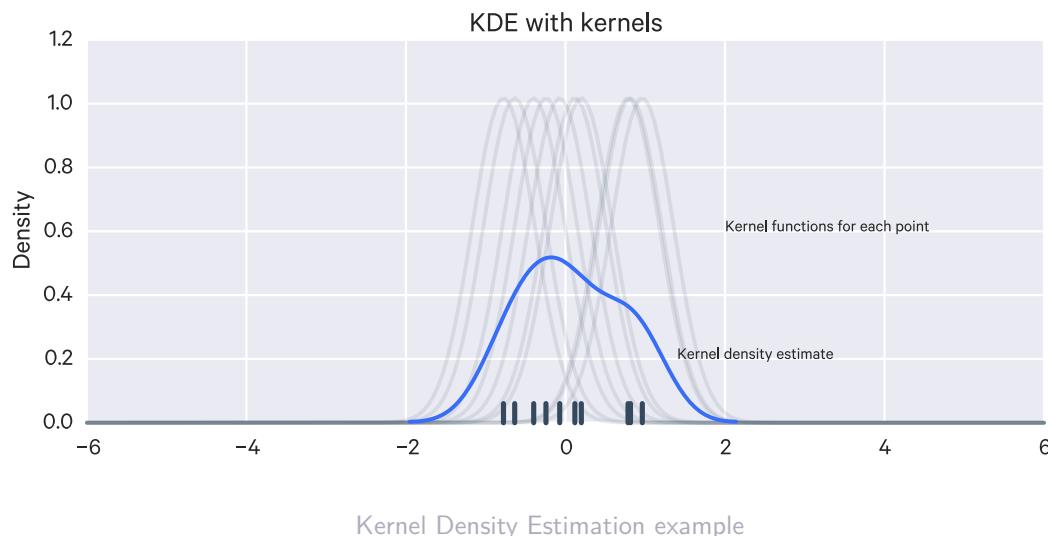
Where a, b are the endpoints of the interval, calculated according to the sampling distribution of Y . We condition on θ because, as just mentioned, in frequentist statistics, the parameters are fixed and the data Y is random.

We can compute the 95% confidence interval by taking the point estimate (which is the best estimate for the value) and $\pm 2SE$, that is build an interval within two standard errors of the point estimate. The interval we add or subtract to the point estimate (here it is $2SE$) is called the **margin of error**. The value we multiply the SE with is essentially a Z score, so we can more generally describe the margin of error as zSE .

For the confidence interval of the mean, we can be more precise and look within $\pm 1.96SE$ (that is, $z = 1.96$) of the point estimate \bar{x} for the 95% confidence interval (that is, our 95% confidence interval would be $\bar{x} \pm 1.96SE$). This is because we know to the sampling distribution for the sample means approximates a normal distribution (for sufficiently large sample sizes, $n \geq 30$ is a rule of thumb) according to the central limit theorem.

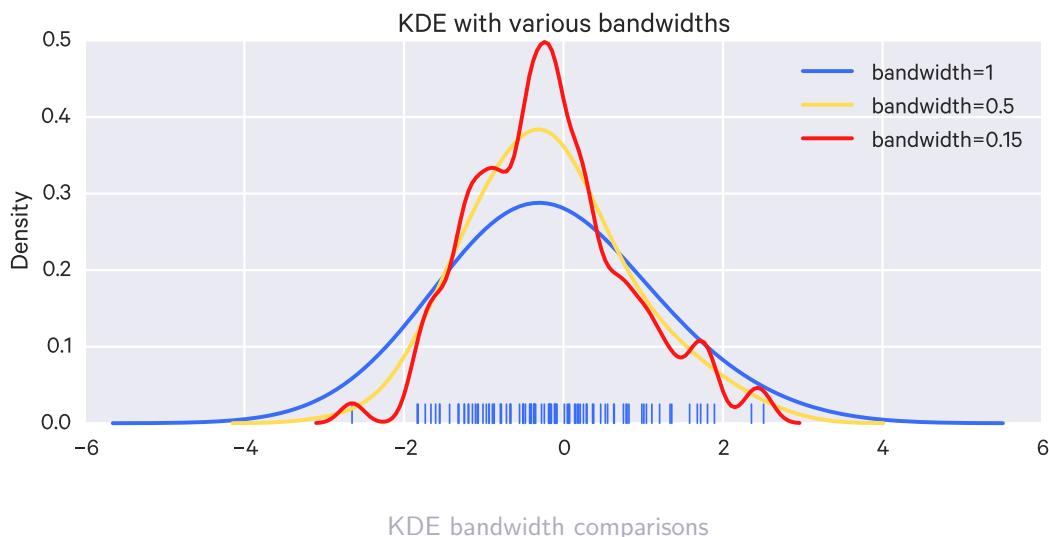
5.2.6 Kernel Density Estimates

Sometimes we don't want the parameters of our data's distribution, but just a smoothed representation of it. **Kernel density estimation** allows us to get this representation. It is a *nonparametric* method because it makes no assumptions about the form of the underlying distribution (i.e. no assumptions about its parameters).



Some kernel function (which generates symmetric densities) is applied to each data point, then the density estimate is formed by summing the densities. The kernel function determines the shape of these densities and the *bandwidth* parameter, $h > 0$, determines their spread and the *smoothing* of the estimate. Typically, a Gaussian kernel function is used, so the bandwidth is equivalent to the variance.

In this figure, the grey curve is the true density, the red curve is the KDE with $h = 0.05$, the black curve is the KDE with $h = 0.337$, and the green curve is the KDE with $h = 2$.



5.3 Experimental Statistics

Experimental statistics is concerned with **hypothesis testing**, where you have a hypothesis and want to learn if your data supports it. That is, you have some sample data and an apparent effect, and you want to know if there is any reason to believe that the effect is genuine and not just by chance.

Often you are comparing two or more groups; more specifically, you are typically comparing statistics across these groups, such as their means. For example, you want to see if the difference of their means is **statistically significant**; which is to say, likely that it is a real effect and not just chance.

The “classical” approach to hypothesis testing, **null hypothesis significance testing (NHST)**, follows this general structure:

1. Quantify the size of the apparent effect by choosing some **test statistic**, which is just a summary statistic which is useful for hypothesis testing or identifying p-values. For example, if you have two populations you’re looking at, this could be the difference in means (of whatever you are measuring) between the two groups.
2. Define a **null hypothesis**, which is usually that the apparent effect is not real.
3. Compute a **p-value**, which is the probability of seeing the effect if the null hypothesis is true.
4. Determine the **statistical significance** of the result. The lower the p-value, the more significant the result is, since the less likely it is to have just occurred by chance.

Broadly speaking, there are two types of scientific studies: **observational** and **experimental**.

In observational studies, the research cannot interfere while recording data; as the name implies, the involvement is merely as an observer.

Experimental studies, however, are deliberately structured and executed. They must be designed to minimize error, both at a low level (e.g. imprecise instruments or measurements) and at a high-level (e.g. researcher biases).

5.3.1 Statistical Power

The power of a study is the likelihood that it will distinguish an effect of a certain size from pure luck. - [Statistical power and underpowered statistics](#), Alex Reinhart

Statistical **power**, sometimes called **sensitivity**, can be defined as the probability of rejecting the null hypothesis when it is false.

If β is the probability of a type II error (i.e. failing to reject the null hypothesis when it's false), then power = $1 - \beta$.

Power...

- Increases as n (sample size) increases
- Increases as σ decreases (less variability)
- Is higher for a one-sided test than for its associated two-sided test

5.3.2 Sample Selection

Bias can enter studies primarily in two ways:

- in the process of selecting the objects to study (sampling and retention)
- in the process of collection information *about* the objects

To prevent selection bias (selecting samples in such a way that it encourages a particular outcome, whether done consciously or not), sample selection may be random.

In the case of medical trials and similar studies, random allocation is ideally **double blind**, so that neither the patient nor the researchers know which treatment a patient is receiving.

Another sample selection technique is **stratified sampling**, in which the population is divided into categories (e.g. male and female) and samples are selected from those subgroups. If the variable used for stratification is strongly related to the variable being studied, there may be better accuracy from the sample size.

You need large sample sizes because with small sample sizes, you're more sensitive to the effects of chance. e.g. if I flip a coin 10 times, it's feasible that I get heads 6/10 times (60% of the time). With that result I couldn't conclusively say whether or not that coin is rigged. If I flip that coin 1000 times, it's extremely unlikely that I will get heads 60% of the time (600/1000 times) if it were a fair coin.

Sometimes to increase sample size, a researcher may use a technique called "replication", which is simply repeating the measurements with new samples. but some researchers really only "pseudoreplicate". samples should be as independent from each other as possible - otherwise you have too many confounding factors. in medical research, researchers may sample a single patient multiple times, every week for instance, and treat each week's sample as a distinct sample. this is pseudoreplication - you begin to inflate other factors particular to that patient in your results. another example is - say you wanted to measure pH levels in soil samples across the US. well, you cant sample soil 15ft from each other because they are too dependent on each other:

Operationalization

Operationalization is the practice of coming up with some way of measuring something which cannot be directly measured, such as intelligence. This may be accomplished via *proxy measurements*.

References

- [Statistics Done Wrong](#), Alex Reinhart

5.3.3 The Null Hypothesis

In an experiment, the **null hypothesis**, notated H_0 , is the “status quo”. For example, in testing whether or not a drug has an impact on a disease, the null hypothesis would be that the drug has no effect.

When running an experiment, you do it under the assumption that the null hypothesis is true. Then you ask: what's the probability of getting the results you got, assuming the null hypothesis is true? If that probability is very small, the null hypothesis is likely false. This probability - of getting your results if the null hypothesis were true - is called the **P value**.

5.3.4 Type 1 Errors

A **type 1 error** is one where the null hypothesis is rejected, even though it is true.

Type 1 errors are usually presented as a probability of them occurring, e.g. a “0.5% chance of a type 1 error” or a “type 1 error with probability of 0.01”.

5.3.5 P Values

P values are central to null hypothesis significance testing (NHST), but they are commonly misunderstood.

P values *do not*:

- tell you the probability of the null hypothesis being true
- tell you the probability of *any* hypothesis being true
- can never prove or disprove hypotheses

There's no mathematical tool to tell you if your hypothesis is true; you can only see whether it is consistent with the data, and if the data is sparse or unclear, your conclusions are uncertain. - [Statistics Done Wrong](#), Alex Reinhart

So what is it then? The P value is the probability of seeing your results or data if the null hypothesis were true.

That is, given data D and a hypothesis H , where H_0 is the null hypothesis, the P value is merely:

$$P(D|H_0)$$

If instead we want to find the probability of our hypothesis given the data, that is, $P(H|D)$, we have to use Bayesian inference instead:

$$P(H|D) = \frac{P(D|H)P(H)}{P(D|H)P(H) + P(D|\neg H)P(\neg H)}$$

Note that P values are problematic when testing multiple hypotheses (**multiple testing or multiple comparisons**) because any “significant” results (as determined by P value comparisons, e.g. $p < 0.5$) may be deceptively so, since that result may still have just been chance, as the following comic illustrates. That is, the more significance tests you conduct, the more likely you will make a Type 1 Error.

In this comic, 20 hypotheses are tested, so with a significance level at 5%, it’s expected that at least one of those tests will come out significant by chance. In the real world this may be problematic in that multiple research groups may be testing the same hypothesis and chances may be such that one of them gets significant results.

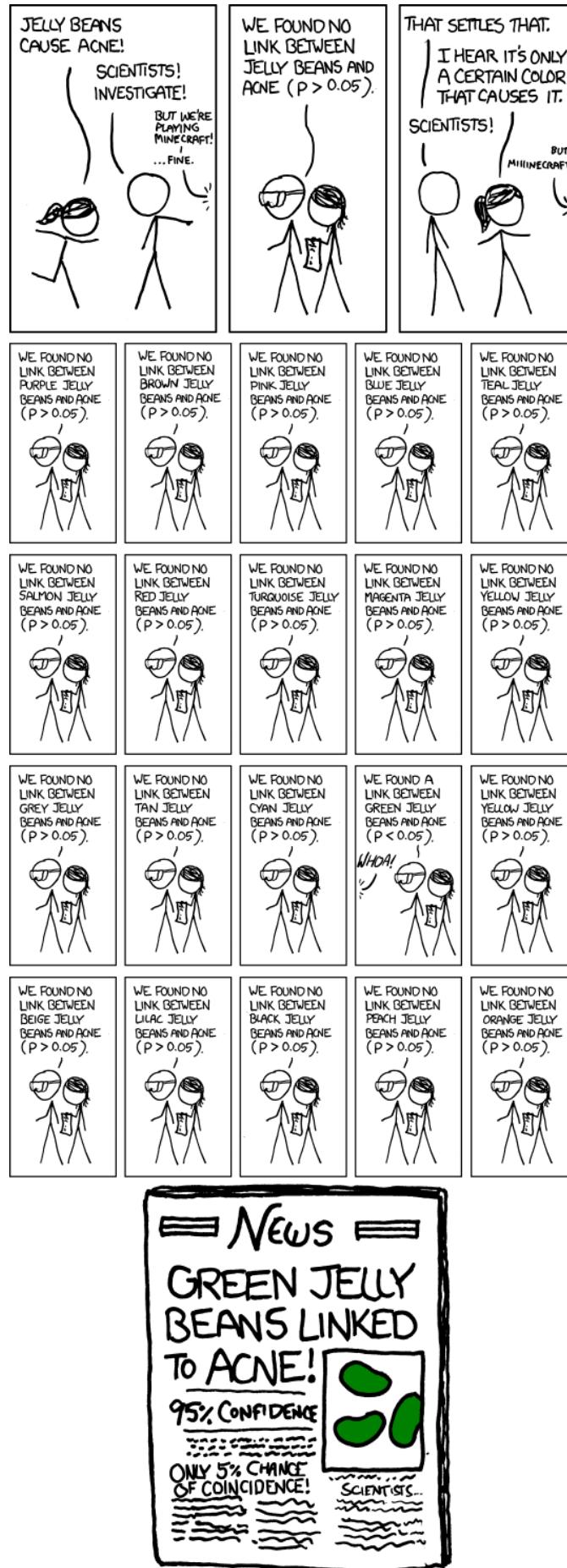
5.3.6 The Base Rate Fallacy

A very important shortcoming to be aware of is the base rate fallacy. A P value cannot be considered in isolation. The base rate of whatever occurrence you are looking at must also be taken into account. Say you are testing 100 treatments for a disease, and it’s a very difficult disease to treat, so there’s a low chance (say 1%) that a treatment will actually be successful. This is your base rate. A low base rate means a higher probability of false positives - treatments which, during the course of your testing, may appear to be successful but are in reality not (i.e. their success was a fluke). A good example is the mammogram test example (see [The p value and the base rate fallacy](#)).

A p value is calculated under the assumption that the medication *does not work* and tells us the probability of obtaining the data we did, or data more extreme than it. It does *not* tell us the chance the medication is effective. - [The p value and the base rate fallacy](#), Alex Reinhart

References

- *Toward Evidence-Based Medical Statistics. 1: The P Value Fallacy*, Steven N. Goodman, MD, PhD
- *Misinterpretations of Significance: A Problem Students Share with Their Teachers?*, Heiko Haller & Stefan Krauss



xkcd - "Significant"

5.3.7 False Discovery Rate

The **false discovery rate** is the expected proportion of false positives (Type 1 errors) amongst hypothesis tests.

For example, if we have a maximum FDR of 0.10 and we have 1000 observations which seem to indicate a significant hypothesis, then we can expect 100 of those observations to be false positives.

The **q value** for an individual hypothesis is the minimum FDR at which the test may be called significant.

Say you run multiple comparisons and have the following values:

- m = the total number of hypotheses tested (number of comparisons)
- m_0 = the number of true null hypotheses (H_0)
- $m - m_0$ = the number of true alternative hypotheses (H_i)
- V = the number of false positives (Type 1 errors)
- S = the number of true positives
- T = the number of false negatives (Type 2 errors)
- U = the number of true negatives
- $R = V + S$ = the number of hypotheses declared significant

We can calculate the FDR as:

$$\text{FDR} = E\left[\frac{V}{V+S}\right] = E\left[\frac{V}{R}\right]$$

Note that $\frac{V}{R} = 0$ if $R = 0$.

5.3.8 Alpha Level

The value that you select to compare the p-value to, e.g. 0.5 in the comic, is the **alpha level** $\bar{\alpha}$, also called the **significance level**, of an experiment. Your alpha level should be selected according to the number of tests you'll be conducting in an experiment.

There are some approaches to help adjust the alpha level.

The Bonferroni Correction

The highly conservative **Bonferroni Correction** can be used as a safeguard.

You divide whatever your significance level $\bar{\alpha}$ is by the number of statistical tests t you're doing:

$$\alpha_p = \frac{\bar{\alpha}}{t}$$

α_p is the per-comparison significance level which you apply for each individual test, and $\bar{\alpha}$ is the maximum experiment-wide significance level, called the **maximum familywise error rate** (FWER).

The Sidak Correction

A more sensitive correction, the **Sidak Correction**, can also be used:

$$\alpha_p = 1 - (1 - \bar{\alpha})^{\frac{1}{n}}$$

For n independent comparisons, α , the experiment-wide significance level (the FWER) is:

$$\alpha = 1 - (1 - \alpha_p)^n$$

For n dependent comparisons, use:

$$\alpha \leq n\alpha_p$$

5.3.9 The Benjamini-Hochberg Procedure

Approaches like the Bonferroni correction lowers the alpha level which end up decreasing your statistical power - that is, you fail to detect false effects *as well as* true effects.

And with such an approach, you are still susceptible to the base rate fallacy, and may still have false positives. So how can you calculate the false discovery rate? That is, what fraction of the statistically Significant results are false positives?

You can use the Benjamini-Hochberg procedure, which tells you which P values to consider statistically significant:

1. Perform your statistical tests and get the P value for each. Make a list and sort it in ascending order.
2. Choose a false-discovery rate q . The number of statistical tests is m .
3. Find the largest p value such that $p \leq \frac{iq}{m}$, where i is the P value's place in the sorted list.
4. Call that P value and all smaller than it statistically significant.

The procedure guarantees that out of all statistically significant results, no more than q percent will be false positives.

The Benjamini-Hochberg procedure is fast and effective, and it has been widely adopted by statisticians and scientists in certain fields. It usually provides better statistical power than the Bonferroni correction and friends while giving more intuitive results. It can be applied in many different situations, and variations on the procedure provide better statistical power when testing certain kinds of data.

Of course, it's not perfect. In certain strange situations, the Benjamini-Hochberg procedure gives silly results, and it has been mathematically shown that it is always possible

to beat it in controlling the false discovery rate. But it's a start, and it's much better than nothing.

Reference: [Controlling the false discovery rate](#), Alex Reinhart

5.3.10 Sum of Squares

The sum of squares within (SSW)

$$\text{SSW} = \sum_{i=1}^m (\sum_{j=1}^n (x_{ij} - \bar{x}_i)^2)$$

- This shows how much of SST is due to variation *within* each group, i.e. variation from within that group's mean.
- The degrees of freedom here is calculated $m(n - 1)$.

The sum of squares between (SSB)

$$\text{SSB} = \sum_{i=1}^m [n_m ((\bar{x}_m - \bar{\bar{x}})^2)]$$

- This shows how much of SST is due to variation between the group means
- The degrees of freedom here is calculated $m - 1$.

The total sum of squares (SST)

$$\text{SST} = \sum_{i=1}^m (\sum_{j=1}^n (x_{ij} - \bar{\bar{x}})^2)$$

$$\text{SST} = \text{SSW} + \text{SSB}$$

- Note: $\bar{\bar{x}}$ is the mean of means, or the "grand mean".
- This is the total variation for the groups
- The degrees of freedom here is calculated $mn - 1$.

5.3.11 Statistical Tests

Two-sided tests

Asks "What is the chance of seeing an effect as big as the observed effect, without regard to its sign?" That is, you are looking for any effect, increase or decrease.

One-sided tests

Asks “What is the chance of seeing an effect as big as the observed effect, with the same sign?” That is, you are looking for either only an increase or decrease.

Unpaired t-test

The most basic statistical test, used when comparing the means from two groups. Used for small sample sizes. The t-test returns a p-value.

Paired t-test

The paired t-test is a t-test used when each datapoint in one group corresponds to one datapoint in the other group.

Chi-squared test

When comparing proportions of two populations, it is common to use the chi-squared statistic:

$$\chi^2 = \sum_i \frac{(O_i - E_i)^2}{E_i}$$

Where O_i is the observed frequencies and E_i is the expected frequencies.

Say for example you want to test if a coin is fair. You expect that, if it is fair, you should see about 50/50 heads and tails - this describes your expected frequencies. You flip the coin and observe the actual resulting frequencies - these are your observed frequencies.

The Chi squared test allows you to determine if these frequencies differ significantly.

ANOVA (Analysis of Variance)

ANOVA, ANCOVA, MANOVA, and MANCOVA are various ways of comparing different groups.

- ANOVA - group A is given a placebo and group B is given the actual medication and the outcome variable to compare is how many pounds were lost
- ANCOVA - same as ANOVA but now there is an additional covariate we consider, e.g. hours of exercise per day
- MANOVA and MANCOVA are multivariate counterparts to the above, for instance we may consider cholesterol levels in addition to weight loss

ANOVA is used to compare three or more groups. It uses a single test to compare the means across multiple groups simultaneously, which avoids using multiple tests to make multiple comparisons (which can lead to differences across groups resulting from chance).

There are a few requirements:

- the observations are independent within and across groups
- the data within each group are nearly normal
- the variance in the groups are about equal across groups

ANOVA tests the null hypothesis that the means across groups are the same (that is, that $\mu_1 = \dots = \mu_k$, if there are k groups), with the alternate hypothesis being that at least one mean is different. We look at the variability in the sample means and see if it is so large that it is unlikely to have been due to chance. The variability we use is the **mean square between groups** (MSG) which has degrees of freedom $df_G = k - 1$.

The MSG is calculated:

$$MSG = \frac{1}{df_G} SSG = \frac{1}{k-1} \sum_{i=1}^k n_i (\bar{x}_i - \bar{x})^2$$

Where the SSG is the **sum of squares between groups** and n_i is the sample size of group i out of k total groups. \bar{x} is the mean of outcomes across all groups.

We need a value to compare the MSG to, which is the **mean square error** (MSE), which measures the variability within groups and has degrees of freedom $df_E = n - k$.

The MSE is calculated:

$$MSE = \frac{1}{df_E} SSE$$

Where the SSE is the **sum of squared errors** and is computed as:

$$SSE = SST - SSG$$

Where the SSG is same as before and the SST is the **sum of squares total**:

$$\begin{aligned} SST &= \sum_{i=1}^n (x_i - \bar{x})^2 \\ SSG &= \sum_{i=1}^k n_i (\bar{x}_i - \bar{x})^2 \end{aligned}$$

ANOVA uses a test statistic called F , which is computed:

$$F = \frac{MSG}{MSE}$$

When the null hypothesis is true, difference in variability across sample means should be due only to chance, so we expect MSG and MSE to be about equal (and thus F to be close to 1).

We take this F statistic and use it with a test called the **F test**, where we compute a p-value from the F statistic, using the F distribution, which has the parameters df_1 and df_2 . We expect ANOVA's F statistic to follow an F distribution with parameters $df_1 = df_G$, $df_2 = df_E$ if the null hypothesis is true.

One-Way ANOVA

Similar to a t-test but used to compare three or more groups. With ANOVA, you calculate the **F statistic**, assuming the null hypothesis²:

$$F = \frac{\frac{SSB}{m-1}}{\frac{SSW}{m(n-1)}}$$

Two-Way ANOVA

Allows you to compare the means of two or more groups when there are multiple variables or factors to be considered.

One-tailed & two-tailed tests

In a **two-tailed test**, both tails of a distribution are considered. For example, with a drug where you're looking for *any* effect, positive or negative.

In a **one-tailed**, only one tail is considered. For example, you may be looking only for a positive or only for a negative effect.

5.3.12 Effect Size

A big part of statistical inference is measuring *effect size*, which more generally is trying to quantify differences between groups, but typically just referred to as "effect size".

There are a few ways of measuring effect size:

Difference in means

The difference in means, e.g. $\mu_1 - \mu_2$

But this has a few problems:

- Must be expressed in the units of measure of the mean (e.g. ft, kg, etc), so it can be difficult to compare to other studies
- Needs more context about the distributions (e.g. standard deviation) to understand if the difference is large or not

²Remember that SSB is the "sum of squares between" and SSW is the "sum of squares within".

Distribution overlap

The overlap between the two distributions:

Choose some threshold between the two means, e.g.

- The midpoint between the means: $\frac{\mu_1 + \mu_2}{2}$
- Where the PDFs cross: $\frac{\sigma_1\mu_1 + \sigma_2\mu_2}{\sigma_1 + \sigma_2}$

Count how many in the first group are below the threshold, call it m_1 . Count how many in the second group are above the threshold, call it m_2 .

The overlap then is:

$$\frac{m_1}{n_1} + \frac{m_2}{n_2}$$

Where n_1, n_2 are the sample sizes of the first and second groups, respectively.

This overlap can also be framed as a *misclassification rate*, which is just $\frac{\text{overlap}}{2}$.

These measures are unitless, which makes them easy to compare across studies.

Probability of superiority

The “probability of superiority” is the probability that a randomly chosen datapoint from group 1 is greater than a randomly chosen datapoint from group 2.

This measure is also unitless.

Cohen's d

Cohen's d is the difference in means, divided by the standard deviation, which is computed from the pooled variance, σ_p^2 , of the groups:

$$\begin{aligned}\sigma_p^2 &= \frac{n_1\sigma_1^2 + n_2\sigma_2^2}{n_1 + n_2} \\ d &= \frac{\mu_1 - \mu_2}{\sqrt{\sigma_p^2}}\end{aligned}$$

This measure is also unitless.

Different fields have different intuitions about how big a d value is; it's something you have to learn.

5.3.13 Reliability

Reliability refers to how consistent or repeatable a measurement is (for continuous data).

There are three main approaches:

Multiple-occasions reliability

Aka *test-retest reliability*. This is how a test holds up over repeated testing, e.g. “temporal stability”. This assumes the underlying metric does not change.

Multiple-forms reliability

Aka *parallel-forms reliability*. This asks: how consistent are different tests at measuring the same thing?

Internal consistency reliability

This asks: do the items on a test all measure the same thing?

5.3.14 Agreement

Agreement is similar to reliability, but used more for discrete data.

Percent agreement

$$\frac{\text{number of cases where tests agreed}}{\text{all cases}}$$

Note that a high percent agreement may be obtained by chance.

Cohen's kappa

Often just called kappa, this corrects for the possibility of chance agreement:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

Where p_o is the *observed agreement*, that is, $\frac{\text{num. agreement}}{\text{total cases}}$, and p_e is the *expected agreement*. Kappa ranges from -1 to 1, where 1 is perfect agreement.

5.4 Handling Data

5.4.1 Transforming data

Occasionally you may find data easier to work with if you apply a **transformation** to it; that is, rescale it in some way. For instance, you might take the natural log of your values, or the square root, or the inverse. This can reduce skew and the effect of outliers or make linear modeling easier.

The function which applies this transformation is called a **link function**.

5.4.2 Dealing with missing data

Data can be missing for a few reasons:

- *Missing completely at random* (MCAR) - missing cases are identical to non-missing cases, on average.
- *Missing at random* (MAR) - Missing data depends on measured values, so they can be modeled by other observed variables.
- *Missing not at random* (MNAR) - Missing data depends on unmeasured/unknown variables, so there is no way to account for them.

There are a few strategies for dealing with missing data.

The worst you can do is to ignore the missing data and try to run your analysis, missing data and all (it likely won't and probably shouldn't work).

Alternatively, you can delete all datapoints which have missing data, leaving only complete data points - this is called **complete case analysis**. Complete case analysis makes the most sense with MCAR missing data - you will have a reduction in sample size, and thus a reduction in statistical power, as a result, but your inference will not be biased. The possibly systemic nature of missing data in MAR and MNAR means that complete case analysis may overlook important details for your model.

You also have the option of filling in missing values - this is called **imputation** (you "impute" the missing values). You can, for instance, filling in missing values with the mean of that variable. You don't gain any of the information that was missing, and you end up ignoring the uncertainty associated with the fill-in value (and the resulting variances will be *artificially reduced*), but you at least get to maintain your sample size. Again, bias may be introduced in MAR and MNAR situations since the missing data may be due to some systemic cause.

One of the better approaches is **multiple imputation**, which produces unbiased parameter estimates and accounts for the uncertainty of imputed values. A regression model is used to generated the imputed values, and does well especially under MAR conditions - the regression model may be able to exploit info in the dataset about the missing data. If some known values correlate with the missing values, they can be of use in this way.

Then, instead of using the regression model to produce one value for each missing value, multiple values are produced, so that the end result is multiple copies of your dataset, each with different imputed values for the missing values. You perform your analysis across all datasets and average the produced estimates.

5.4.3 Resampling

Resampling involves repeatedly drawing subsamples from an existing sample.

Resampling is useful for assessing and selecting models and for estimating the precision of parameter estimates.

A common resampling method is **bootstrapping**.

Bootstrapping

Bootstrapping is a resampling method to approximate the true sampling distribution of a dataset, which can then be used to estimate the mean and the variance of the distribution. The advantage with bootstrapping is that there is no need to compute derivatives or make assumptions about the distribution's form.

You take R samples S_i^* , with replacement, each of size n (i.e. each resample is the same size as the original sample), from your dataset. These samples, $S^* =$

S_1^*, \dots, S_R^* are called *replicate* bootstrap samples. Then you can compute an estimate of the t statistic for each of the bootstrap samples, $T_i^* = t(S_i^*)$.

Then you can estimate the mean and variance:

$$\bar{T}^* = \hat{E}[T^*] = \frac{\sum_i T_i^*}{R}$$

$$\hat{\text{Var}}(T^*) = \frac{\sum_i (T_i^* - \bar{T}^*)^2}{R - 1}$$

With bootstrap estimates, there are two possible sources of error. You may have the **sampling error** from your original sample S in addition to the **bootstrap error**, from failing to be comprehensive in your sampling of bootstrap samples. To avoid the latter, you should try to choose a large R , such as $R = 1000$.

5.5 References

- IFT 725 Review of Fundamentals
- [Statistical Inference Course Notes](#), Xing Su
- [Regression Models Course Notes](#), Xing Su
- Statistics in a Nutshell
- <https://stats.stackexchange.com/questions/71962/what-is-the-difference-between-descriptive-and-inferential-statistics>
- GYLO Statistics 1, version 1.5, lesson 1.6
- <http://www.countbayesie.com/blog/2015/2/21/variance-co-variance-and-correlation>
- Think Stats: Exploratory Data Analysis in Python, Version 2.0.27. Allen B Downey.
- *Principles of Statistics*, M.G. Bulmer. 1979.
- OpenIntro Statistics, Second Edition. David M Diez, Christopher D Barr, Mine Çetinkaya-Rundel.
- Computational Statistics I. Allen Downey. SciPy 2015: <https://sites.google.com/site/pyinference/home/scipy-2015>
- Computational Statistics II. Chris Fonnesbeck. SciPy 2015: <https://www.youtube.com/watch?v=heFaYLKVZY4> and https://github.com/fonnesbeck/scipy2015_tutorial
- Bayesian Statistical Analysis. Chris Fonnesbeck. SciPy 2014: https://github.com/fonnesbeck/scipy2014_tutorial
- Lecture Notes from CS229 (Stanford): <http://cs229.stanford.edu/materials.html>

- *Data Analysis Using Regression and Multilevel/Hierarchical Models*, 1st edition. Andrew Gelman and Jennifer Hill.
- <http://jakevdp.github.io/blog/2014/03/11/frequentism-and-bayesianism-a-practical-intro/>
- Andrew Ng's Coursera *Machine Learning* course (2014)
- Introduction to Artificial Intelligence (Udacity CS271): <https://www.udacity.com/wiki/cs271>, Peter Norvig and Sebastian Thrun.

6

Bayesian Statistics

Bayesian statistics (also called **Bayesian inference**) is an approach to statistics contrasted with frequentist approaches.

As is with frequentist statistical inference, Bayesian inference is concerned with estimating parameters from some observed data. However, whereas frequentist inference returns point estimates - that is, single values - for these parameters, Bayesian inference instead expresses these parameters themselves as probability distributions. This is intuitively appealing as we are *uncertain* about the parameters we've inferred; with Bayesian inference we can represent this uncertainty.

This is to say that in Bayesian inference, we don't assign an explicit value to an unknown parameter. Rather, we define it over a probability distribution as well: what values is the parameter *likely* to take on? That is, we treat the parameter itself as a random variable.

We may say for instance that an unknown parameter θ is drawn from an exponential distribution:

$$\theta \sim \text{Exp}(\alpha)$$

Here α is a *hyperparameter*, that is, it is a parameter for our parameter θ .

Fundamentally, this is Bayesian inference:

$$P(\theta|X)$$

Where the parameters θ are the unknown, so we express them as a probability distribution, given the observations X . This probability distribution is the **posterior distribution**.

So we must decide (specify) probability distributions for both the data sample and for the unknown parameters. These decisions involve making a lot of assumptions. Then you must compute a posterior distribution, which often cannot be calculated analytically - so other methods are used (such as simulations, described later).

From the posterior distribution, you can calculate point estimates, credible intervals, quantiles, and make predictions.

Finally, because of the assumptions which go into specifying the initial distributions, you must test your model and see if it fits the data and seems reasonable.

Thus Bayesian inference amounts to:

1. Specifying a sampling model for the observed data X , conditioned on the unknown parameter θ (which we treat as a random variable), such that $X \sim f(X|\theta)$, where $f(X|\theta)$ is either the PDF or the PMF (as appropriate).
2. Specifying a marginal or distribution $\pi(\theta)$ for θ , which is the prior distribution ("prior" for short): $\theta \sim \pi(\theta)$
3. From this we wish to compute the posterior, that is, uncover the distribution for θ given the observed data X , like so: $\pi(\theta|X) = \frac{\pi(\theta)L(\theta|X)}{\int \pi(\theta)L(\theta|X)d\theta}$, where $L(\theta|X) \propto f(\theta|X)$ in θ , called the likelihood of θ given X . More often than not, the posterior must be approximated through Markov Chain Monte Carlo (detailed later).

6.0.1 Frequentist vs Bayesian approaches

For frequentists, probability is thought of in terms of frequencies, i.e. the probability of the event is the amount of times it happened over the total amount of times it could have happened.

In frequentist statistics, the observed data is considered random; if you gathered more observations they would be different according to the underlying distribution. The parameters of the model, however, are considered fixed.

For Bayesians, probability is belief or certainty about an event. Observed data is considered fixed, but the model parameters are random (uncertain) instead and considered to be drawn from some probability distribution.

Another way of phrasing this is that frequentists are concerned with uncertainty in the data, whereas Bayesians are concerned with uncertainty in the parameters.

6.1 Bayes' Rule

In frequentist statistics, many different estimators may be used, but in Bayesian statistics the only estimator is Bayes' Formula (aka Bayes' Rule or Bayes' Theorem).

Bayes' Theorem, aka Bayes' Rule:

- H is the hypothesis (more commonly represented as the parameters θ)
- D is the data

$$P(H|D) = \frac{P(H)P(D|H)}{P(D)}$$

- $P(H)$ = the probability of the hypothesis before seeing the data. The *prior*.
- $P(H|D)$ = probability of the hypothesis, given the data. The *posterior*.
- $P(D|H)$ = the probability of the data under the hypothesis. The *likelihood*.
- $P(D)$ = the probability of data under *any* hypothesis. The *normalizing constant*.

For an example of likelihood:

If I want to predict the sides of a dice I rolled, and then I rolled an 8, then $P(D|\text{a six sided die}) = 0$. That is, it is impossible to have my observed data under the hypothesis of having a six sided die.

A key insight to draw from Bayes' Rule is that $P(H|D) \propto P(H)P(D|H)$, that is, the posterior is proportional to the product of the prior and the likelihood.

Note that the normalizing constant $P(D)$ usually cannot be directly computed and is equivalent to $\int P(D|H)P(H)dH$ (which is usually intractable since there are usually multiple parameters of interest, resulting in a multidimensional integration problem. If θ , the parameters, is one dimensional, then you could integrate it rather easily).

One workaround is to do approximate inference with non-normalized posteriors, since we know that the posterior is proportional to the numerator term:

$$P(H|D) \propto P(H)P(D|H)$$

Another workaround to approximate the posterior using simulation methods such as Monte Carlo.

Given a set of hypotheses H_0, H_1, \dots, H_n , the distribution for the priors of these hypotheses is the *prior distribution*, i.e. $P(H_0), P(H_1), \dots, P(H_n)$.

The distribution of the posterior probabilities is the *posterior distribution*, i.e. $P(H_0|D), P(H_1|D), \dots, P(H_n|D)$.

6.2 Choosing a prior distribution

With Bayesian inference, we must *choose* a prior distribution, then apply data to get our posterior distribution. The prior is chosen based on domain knowledge or intuition or perhaps from the results of previous analysis; that is, it is chosen subjectively - there is no prescribed formula for picking a prior. If you have no idea what to pick, you can just pick a uniform distribution as your prior.

Your choice of prior will affect the posterior that you get, and the subjectivity of this choice is what makes Bayesian statistics controversial - but it's worth noting that all of statistics, whether or frequentist or Bayesian, involves many subjective decisions (e.g. frequentists must decide on an estimator to use, what data to collect and how, and so on) - what matters most is that you are explicit about your decisions and why you made them.

Say we perform an Bayesian analysis and get a posterior. Then we get some new data for the same problem. We can re-use the posterior from before as our prior, and when we run Bayesian analysis on the new data, we will get a new posterior which reflects the additional data. We don't have to re-do any analysis on the data from before, all we need is the posterior generated from it.

For any unknown quantity we want to model, we say it is drawn from some prior of our choosing. This is usually some parameter describing a probability distribution, but it could be other values as well. This is central to Bayesian statistics - all unknowns are represented as distributions of possible values. In Bayesian statistics: if there's a value and you don't know what it is, come up with a prior for it and add it to your model!

If you think of distributions as landscapes or surfaces, then the data deforms the prior surface to mold it into the posterior distribution.

The surface's "resistance" to this shaping process depends on the selected prior distribution.

When it comes to selecting Bayesian priors, there are two broad categories:

- *objective priors* - these let the data influence the posterior the most
- *subjective priors* - these allow the practitioner to asset their own views in to the prior. This prior can be the posterior from another problem or just come from domain knowledge.

An example objective prior is a *uniform* (flat) prior where every value has equal weighting. Using a uniform prior is called *The Principle of Indifference*. Note that a uniform prior restricted within a range is *not* objective - it has to be over *all* possibilities.

Note that the more data you have (as N increases), the choice of prior becomes less important.

6.2.1 Conjugate priors

Conjugate priors are priors which, when combined with the likelihood, result in a posterior which is in the same family. These are very convenient because the posterior can be calculated analytically, so there is no need to use approximation such as Markov Chain Monte Carlo (see below).

For example, a binomial likelihood is a conjugate with a beta prior - their combination results in a beta-binomial posterior.

For example, the Gaussian family of distributions are conjugate to itself (*self conjugate*) - a Gaussian likelihood with a Gaussian prior results in a Gaussian posterior.

For example, when working with count data you will probably use the Poisson distribution for your likelihood, which is conjugate with gamma distribution priors, resulting in a gamma posterior.

Unfortunately, conjugate priors only really show up in simple one-dimensional models.

More generally, we can define a conjugate prior like so:

Say random variable X comes from a well-known distribution, f_α where α are the possibly unknown parameters of f . It could be a normal, binomial, etc distribution.

For the given distribution f_α , there may exist a prior distribution p_β such that

$$\widehat{p_\beta} \cdot \widehat{f_\alpha(X)} = \widehat{p_{\beta'}}$$

Beta-Binomial Model

The Beta-Binomial model is a useful Bayesian model because it provides values between 0 and 1, which is useful for estimating probabilities or percentages.

It involves, as you might expect, a beta and a binomial distribution.

So say we have N trials and observe n successes. We describe these observations by a binomial distribution, $n \sim \text{Bin}(N, p)$ for which p is unknown. So we want to come up with some distribution for p (remember, with Bayesian inference, you do not produce point estimates, that is, a single value, but a distribution for your unknown value to describe the uncertainty of its true value).

For frequentist inference we'd estimate $\hat{p} = \frac{n}{N}$ which isn't quite good for low numbers of N .

This being Bayesian inference, we first must select a prior. p is a probability and therefore is bound to $[0, 1]$. So we could choose a uniform prior over that interval; that is $p \sim \text{Uniform}(0, 1)$.

However, $\text{Uniform}(0, 1)$ is equivalent to a beta distribution where $\alpha = 1, \beta = 1$, i.e. $\text{Beta}(1, 1)$. The beta distribution is bound between 0 and 1 so it's a good choice for estimating probabilities.

We prefer a beta prior over a uniform prior because, given binomial observations, the posterior will also be a beta distribution.

It works out nicely mathematically:

$$\begin{aligned} p &\sim \text{Beta}(\alpha, \beta) \\ n &\sim \text{Bin}(N, p) \\ p | n, N &\sim \text{Beta}(\alpha + n, \beta + N - n) \end{aligned}$$

So with these two distributions, we can directly compute the posterior with no need for simulation (e.g. MCMC).

Example

We run 100 trials and observe 10 successes. What is the probability p of a successful trial?

Our knowns are $N = 100, n = 10$. A binomial distribution describes these observations, but we have the unknown parameter p .

For our prior for p we choose $\text{Beta}(1, 1)$ since it is equivalent to a uniform prior over $[0, 1]$ (i.e. it is an objective prior).

We can directly compute the posterior now:

$$\begin{aligned} p | n, N &\sim \text{Beta}(\alpha + n, \beta + N - n) \\ p &\sim \text{Beta}(11, 91) \end{aligned}$$

Then we can draw samples from the distribution and compute its mean or other descriptive statistics such as the credible interval.

6.2.2 Sensitivity Analysis

The strength of the prior affects the posterior - the stronger your prior beliefs, the more difficult it is to change those beliefs (it requires more data/evidence). You can conduct **sensitivity analysis** to try your approach with various different priors to get an idea of how different priors affect your resulting posterior.

6.2.3 Empirical Bayes

Empirical Bayes is a method which combines frequentist and Bayesian approaches by using frequentist methods to select the hyperparameters.

For instance, say you want to estimate the μ parameter for a normal distribution.

You could use the empirical sample mean from the observed data:

$$\mu_p = \frac{1}{N} \sum_{i=0}^N X_i$$

Where μ_p denotes the prior μ .

Though if working with not much data, this kind of ends like double-counting your data.

6.3 Markov Chain Monte Carlo (MCMC)

With Bayesian inference, in order to describe your posterior, you often must evaluate complex multi-dimensional integrals (i.e. from very complex, multidimensional probability distributions), which can be computationally intractable.

Instead you can generate sample points from the posterior distribution and use those samples to compute whatever descriptions you need. This technique is called **Monte Carlo integration**, and the process of drawing repeated random samples in this way is called **Monte Carlo simulation**. In particular, we can use a family of techniques known as **Markov Chain Monte Carlo**, which combine Monte Carlo integration and simulation with Markov chains, to generate samples for us.

6.3.1 Monte Carlo Integration

Monte Carlo integration is a way to approximate complex integrals using random number generation.

Say we have a complex integral:

$$\int h(x) dx$$

If we can decompose $h(x)$ into the product of a function $f(x)$ and a probability density function $P(x)$ describing the probabilities of the inputs x , then:

$$\int h(x)dx = \int f(x)P(x)dx = E_{P(x)}[f(x)]$$

That is, the result of this integral is the expected value of $f(x)$ over the density $P(x)$.

We can approximate this expected value by taking the mean of many, many samples (n samples):

$$\int h(x)dx = E_{P(x)}[f(x)] \approx \frac{1}{n} \sum_{i=1}^n f(x_i)$$

This process of approximating the integral is *Monte Carlo integration*.

For very simple cases of known distributions, we can sample directly, e.g.

```
import numpy as np

# Say we think the distribution is a Poisson distribution
# and the parameter of our distribution, lambda,
# is unknown and what we want to discover.
lam = 5

# Collect 100000 samples
sim_vals = np.random.poisson(lam, size=100000)

# Get whatever descriptions we want, e.g.
mean = sim_vals.mean()

# For poisson, the mean is lambda, so we expect
# them to be approximately equal (given a large enough sample size)
abs(lam - mean()) < 0.001
```

6.3.2 Markov Chains

Markov chains are a stochastic process in which the next state depends only on the current state.

Consider a random variable X and a time index t . The state of X at time t is notated X_t .

For a Markov chain, the state X_{t+1} depends only on the current state X_t , that is:

$$P(X_{t+1} = x_{t+1} | X_t = x_t, X_{t-1} = x_{t-1}, \dots, X_0 = x_0) = P(X_{t+1} = x_{t+1} | X_t = x_t)$$

Where $P(X_{t+1} = x_{t+1})$ is the **transition probability** of $X_{t+1} = x_{t+1}$. The collection of transition probabilities is called a **transition matrix** (for discrete states); more generally it is called a **transition kernel**.

If we consider t going to infinity, the Markov chain settles on a **stationary distribution**, where $P(X_t) = P(X_{t-1})$. The stationary distribution does not depend on the initial state of the network. Markov chains are *ergodic*, i.e. they “mix”, which means that the influence of the initial state weakens with time (the rate at which it mixes is its *mixing speed*).

If we call the $k \times k$ transition matrix P and the marginal probability of a state at time t is a $k \times 1$ vector π , then the distribution of the state at time $t+1$ is $\pi'P$. If $\pi'P = \pi'$, then π is the stationary distribution of the Markov chain.

6.3.3 Markov Chain Monte Carlo

MCMC is useful because often we may encounter distributions which aren't easily expressed mathematically (e.g. their functions may have very strange shapes), but we still want to compute some descriptive statistics (or make other computations) from them. MCMC allows us to work with such distributions without needing precise mathematical formulations of them.

More generally, MCMC is really useful if you don't want to (or can't) find the underlying function describing something. As long as you can simulate that process in some way, you don't need to know the exact function - you can just generate enough sample data to work with instead. So MCMC is a brute force but effective method.

Rather than directly compute the integral for posterior distributions in Bayesian analysis, we can instead use MCMC to draw several (thousands, millions, etc) samples from the probability distribution, then use these samples to compute whatever descriptions we'd like about the distribution (often this is some expected value of a function, $E[f(x)]$, where its inputs are drawn from distribution, i.e. $x \sim p$, where p is some probability distribution).

You start with some random initial sample and, based on that sample, you pick a new sample. This is the Markov Chain aspect of MCMC - the next sample you choose depends only on the current sample. This works out so that you spend most your time with high probability samples (b/c they have higher transition probabilities) but occasionally jump out to lower probability samples. Eventually the MCMC chain will converge on a random sample.

So we can take all these N samples and, for example, compute the expected value:

$$E[f(x)] \approx \frac{1}{N} \sum_{i=1}^N f(x_i)$$

Because of the random initialization, there is a “**burn-in**” phase in which the sampling model needs to be “warmed up” until it reaches an equilibrium sampling state, the *stationary distribution*. So you discard the first hundred or thousand or so samples as part of this burn-in phase. You can (eventually) arrive at this stationary distribution *independent of where you started* which is why the random initialization is ok - this is an important feature of Markov Chains.

MCMC is a general technique of which there are several algorithms.

Rejection Sampling

Monte Carlo integration allows us to draw samples from a posterior distribution with a known parametric form. It does not, however, enable us to draw samples from a posterior distribution without a known parametric form. We may instead use **rejection sampling** in such cases.

We can take our function $f(x)$ and if it has bounded/finite *support* ("support" is the x values where $f(x)$ is non-zero, and can be thought of the range of meaningful x values for $f(x)$), we can calculate its maximum and then define a bounding rectangle with it, encompassing all of the support values. This envelope function should contain all possible values of $f(x)$. Then we can randomly generate points from within this box and check if they are under the curve (that is, less than $f(x)$ for the point's x value). If a point is not under the curve, we reject it. Thus we approximate the integral like so:

$$\frac{\text{points under curve}}{\text{points generated}} \times \text{box area} = \lim_{n \rightarrow \infty} \int_A^B f(x) dx$$

In the case of unbounded support (i.e. infinite tails), we instead choose some *majorizing* or *enveloping* function $g(x)$ ($g(x)$ is typically a probability density itself and is called a *proposal density*) such that $cg(x) \geq f(x), \forall x \in (-\infty, \infty)$, where c is some constant. This functions like the bounding box from before. It completely encloses f . Ideally we choose $g(x)$ so that it is close to the target distribution, that way most of our sampled points can be accepted.

Then, for each x_i we draw (i.e. sample), we also draw a uniform random value u_i . Then if $u_i < \frac{f(x_i)}{cg(x_i)}$, we accept x_i , otherwise, we reject it.

The intuition here is that the probability of a given point being accepted is proportional to the function f at that point, so when there is greater density in f for that point, that point is more likely to be accepted.

In multidimensional cases, you draw candidates from every dimension simultaneously.

Metropolis-Hastings

The **Metropolis-Hastings** algorithm uses Markov chains with rejection sampling.

The proposal density $g(\theta_t)$ is chosen as in rejection sampling, but it depends on θ_{t-1} , i.e. $g(\theta_t | \theta_{t-1})$.

First select some initial θ , θ_1 .

Then for n iterations:

- Draw a candidate $\theta_t^c \sim g(\theta_t | \theta_{t-1})$
- Compute the Metropolis-Hastings ratio: $R = \frac{f(\theta_t^c)g(\theta_{t-1} | \theta_t^c)}{f(\theta_{t-1})g(\theta_t^c | \theta_{t-1})}$
- Draw $u \sim \text{Uniform}$
- If $u < R$, accept $\theta_t = \theta_t^c$, otherwise, $\theta_t = \theta_{t-1}$

There are a few required properties of the Markov chain for this to work properly:

- The stationary distribution of the chain must be the target density:
 - The chain must be *recurrent* - that is, for all $\theta \in \Theta$ in the *target density* (the density we wish to approximate), the probability of returning to any state $\theta_i \in \Theta$ = 1. That is, it must be possible *eventually* for any state in the state space to be reached.
 - The chain must be *non-null* for all $\theta \in \Theta$ in the target density; that is, the expected time to recurrence is finite.
 - The chain must have a stationary distribution equal to the target density.
- The chain must be *ergodic*, that is:
 - The chain must be *irreducible* - that is, any state θ_i can be reached from any other state θ_j in a finite number of transitions (i.e. the chain should not get stuck in any infinite loops)
 - The chain must be *aperiodic* - that is, there should not be a fixed number of transitions to get from any state θ_i to any state θ_j . For instance, it should not always take three steps to get from one place to another - that would be a period. Another way of putting this - there are no fixed cycles in the chain.

It can been proven that the stationary distribution of the Metropolis-Hastings algorithm is the target density (proof omitted).

The ergodic property (whether or not the chain “mixes” well) can be validated with some *convergence diagnostics*. A common method is to plot the chain’s values as their drawn and see if the values tend to concentrate around a constant; if not, you should try a different proposal density.

Alternatively, you can look at an autocorrelation plot, which measures the internal correlation (from -1 to 1) over time, called “lag”. We expect that the greater the lag, the less the points should be autocorrelated - that is, we expect autocorrelation to smoothly decrease to 0 with increasing lag. If autocorrelation remains high, then the chain is not fully exploring the space. Autocorrelation can be improved by *thinning*, which is a technique where only every k th draw is kept and others are discarded.

Finally, you also have the options of running multiple chains, each with different starting values, and combining those samples.

You should also use burn-in.

Gibbs Sampling

It is easy to sample from simple distributions. For example, for a binomial distribution, you can basically just flip a coin. For a multinomial distribution, you can basically just roll a dice.

If you have a multinomial, multivariate distribution, e.g. $P(x_1, x_2, \dots, x_n)$, things get more complicated. If the variables are independent, you can factorize the multivariate distribution as a product of univariate distributions, treating each as a univariate multinomial distribution, i.e. $P(x_1, x_2, \dots, x_n) = P(x_1) \times P(x_2) \times \dots \times P(x_n)$. Then you can just sample from each distribution individually, i.e. as a dice roll.

However - what if these aren't independent, and we want to sample from the *joint distribution* $P(x_1, x_2, \dots, x_n)$? We can't factorize it into simpler distributions like before.

With Gibbs sampling we can approximate this joint distribution under the condition that we can easily sample from the conditional distribution for each variable, i.e. $P(x_i|x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$. (This condition is satisfied on Bayesian networks.)

We take advantage of this and iteratively sample from these conditional distributions and using the most recent value for each of the other variables (starting with random values at first). For example, sampling $x_1|x_2, \dots, x_n$, then fixing this value for x_1 while sampling $x_2|x_1, x_3, \dots, x_n$, then fixing both x_1 and x_2 while sampling $x_3|x_1, x_2, x_4, \dots, x_n$, and so on.

If you iterate through this a large number of times you get an approximation of samples taken from the actual joint distribution.

Another way to look at Gibbs sampling:

Say you have random variables c, r, t (cloudy, raining, thundering) and you have the following probability tables:

| c | $P(c)$ |
|-----|--------|
| 0 | 0.5 |
| 1 | 0.5 |

| c | r | $P(r c)$ |
|-----|-----|----------|
| 0 | 0 | 0.9 |
| 0 | 1 | 0.1 |
| 1 | 0 | 0.1 |
| 1 | 1 | 0.9 |

| c | r | t | $P(t c,r)$ |
|-----|-----|-----|------------|
| 0 | 0 | 0 | 0.9 |
| 0 | 0 | 1 | 0.1 |
| 0 | 1 | 0 | 0.5 |
| 0 | 1 | 1 | 0.5 |
| 1 | 0 | 0 | 0.6 |
| 1 | 0 | 1 | 0.4 |
| 1 | 1 | 0 | 0.1 |
| 1 | 1 | 1 | 0.9 |

We can first pick some starting sample, e.g. $c = 1, r = 0, t = 1$.

Then we fix $r = 0, t = 1$ and randomly pick another c value according to the probabilities in the table (here it is equally likely that we get $c = 0$ or $c = 1$). Say we get $c = 0$. Now we have a new sample $c = 0, r = 0, t = 1$.

Now we fix $c = 0, t = 1$ and randomly pick another r value. Here r is dependent only on c . $c = 0$ so we have a 0.9 probability of picking $r = 0$. Say that we do. We have another sample $c = 0, r = 0, t = 1$, which happens to be the same as the previous sample.

Now we fix $c = 0, r = 0$ and pick a new t value. t is dependent on both c and r . $c = 0, r = 0$, so we have a 0.9 chance of picking $t = 0$. Say that we do. Now we have another sample $c = 0, r = 0, t = 0$. Then we repeat this process until convergence (or for some specified number of iterations).

Your samples will reflect the actual joint distribution of these values, since more likely samples are, well, more likely to be generated.

6.4 Bayesian point estimates

Bayesian inference returns a distribution (the posterior) but we often need a single value (or a vector in multivariate cases). So we choose a value from the posterior. This value is a **Bayesian point estimate**.

Selecting the MAP (*maximum a posterior*) value is insufficient because it neglects the shape of the distribution.

Suppose $P(\theta|X)$ is the posterior distribution of θ after observing data X .

The *expected loss* of choosing estimate $\hat{\theta}$ to estimate θ (the true parameter), also known as the *risk* of estimate $\hat{\theta}$ is:

$$I(\hat{\theta}) = E_{\theta}[L(\theta, \hat{\theta})]$$

Where $L(\theta, \hat{\theta})$ is some loss function.

You can approximate the expected loss using the Law of Large Numbers, which just states that as sample size grows, the expected value approaches the actual value. That is, as N grows, the expected loss approaches 0.

For approximating expected loss, it looks like:

$$\frac{1}{N} \sum_{i=1}^N L(\theta_i, \hat{\theta}) \approx E_{\theta}[L(\theta, \hat{\theta})] = I(\hat{\theta})$$

You want to select the estimate $\hat{\theta}$ which minimizes this expected loss:

$$\operatorname{argmin}_{\hat{\theta}} E_{\theta}[L(\theta, \hat{\theta})]$$

6.5 Credible Intervals (Credible Regions)

In Bayesian statistics, The closest analog to confidence intervals in frequentist statistics is the **credible interval**. It is *much* easier to interpret than the confidence interval because it is exactly what most people confuse the confidence interval to be. For instance, the 95% credible interval is the interval in which we expect to find θ 95% of the time.

Mathematically this is expressed as:

$$P(a(y) < \theta < b(y)|Y = y) = 0.95$$

We condition on Y because in Bayesian statistics, the data is fixed and the parameters are random.

6.6 Bayesian Regression

The Bayesian methodology can be applied to regression as well. In conventional regression the parameters are treated as fixed values that we uncover. In Bayesian regression, the parameters are treated as random variables, as they are elsewhere in Bayesian statistics. We define prior distributions for each parameter - in particular, normal priors, so that for each parameter we define a prior mean as well as a covariance matrix for all the parameters.

So we specify:

- b_0 - a vector of prior means for the parameters
- B_0 - a covariance matrix such that $\sigma^2 B_0$ is the prior covariance matrix of β
- $v_0 > 0$ - the degrees of freedom for the prior
- $\sigma_0^2 > 0$ - the variance for the prior (which essentially functions as your strength of belief in the prior - the lower the variance, the more concentrated your prior is around the mean, thus the stronger your belief)

So the prior for your parameters then is a normal distribution parameterized by (b_0, B_0) .

Then v_0 and σ_0^2 give a prior for σ^2 , which is an inverse gamma distribution parameterized by $(v_0, \sigma_0^2 v_0)$.

Then there are a few formulas:

$$\begin{aligned}
b_1 &= (B_0^{-1} + X'X)^{-1}(B_0^{-1}b_0 + X'\hat{\beta}) \\
B_1 &= (B_0^{-1} + X'X)^{-1} \\
v_1 &= v_0 + n \\
v_1\sigma_1^2 &= v_0\sigma_0^2 + S + r \\
S &= \text{sum of squared errors of the regression} \\
r &= (b_0 - \hat{\beta})'(B_0 + (X'X)^{-1})^{-1}(b_0 - \hat{\beta}) \\
f(\beta \mid \sigma^2, y, x) &= \Phi(b_1, \sigma^2 B_1) \\
f(\sigma^2 \mid y, x) &= \text{inv.gamma}\left(\frac{v_1}{2}, \frac{v_1\sigma_1^2}{2}\right) \\
f(\beta \mid y, x) &= \int f(\beta \mid \sigma^2, y, x)f(\sigma^2 \mid y, x)d\sigma^2 = t(b_1, \sigma_1^2 B_1, \text{degrees of freedom} = v_1)
\end{aligned}$$

So the resulting distribution of parameters is a multivariate t distribution.

6.7 A Bayesian example

Let's say we have a coin. We are uncertain whether or not it's a fair coin. What can we learn about the coin's fairness from a Bayesian approach?

Let's restate the problem. We can represent the outcome of a coin flip with a random variable, X . If the coin is not fair, we expect to see heads 100% of the time. That is, if the coin is unfair, $P(X = \text{heads}) = 1$. Otherwise, we expect it to be around $P(X = \text{heads}) = 0.5$.

It's reasonable to assume that X is drawn from a binomial distribution, so we'll use that. The binomial distribution is parameterized by n , the number of trials, and p , the probability of a "success" (in this case, a heads), on a given flip. We can restate our previous statements about the coin's fairness in terms of this parameter p . That is, if the coin is unfair, we expect $p = 1$, otherwise, we expect it to be around $p = 0.5$.

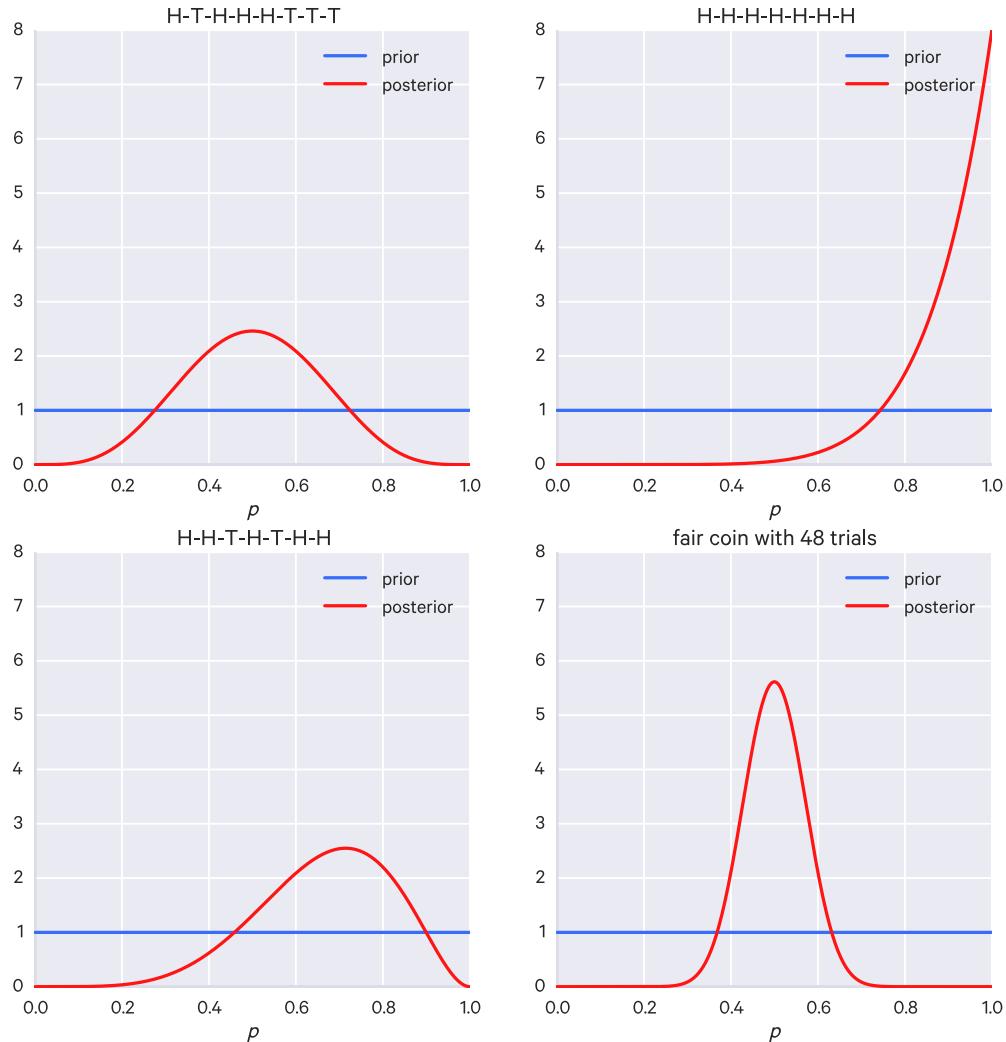
Thus p is the unknown parameter we are interested in, and with the Bayesian approach, we consider it a random variable as well; i.e. drawn from some distribution. First we must state what we believe this distribution to be *prior* to any evidence (i.e. decide on a prior to use). Because p is a probability, the beta distribution seems like a good choice since it is bound to $[0, 1]$ like a probability. The beta distribution has the additional advantage of being a *conjugate prior*, so the posterior is analytically derived and requires no simulation.

The beta distribution is parameterized by α and β (i.e. they are our hyperparameters, $\text{Beta}(\alpha, \beta)$). Here we can choose values for α and β depending on how we choose to proceed. Let's be conservative and use an uninformative prior, that is, a uniform/flat prior, acting as if we don't feel strongly about the coin's bias either way prior to flipping the coin. The beta distribution $\text{Beta}(1, 1)$ is flat.

The posterior for a beta prior will not be derived here, but it is $\text{Beta}(\alpha + k, \beta + (n - k))$, where k is the number of successes (heads) in our evidence, and n is the total number of trials in our evidence.

Now we can flip the coin a few times to gather our evidence.

Below are some illustrations of possible evidence with the prior and the resulting posterior.



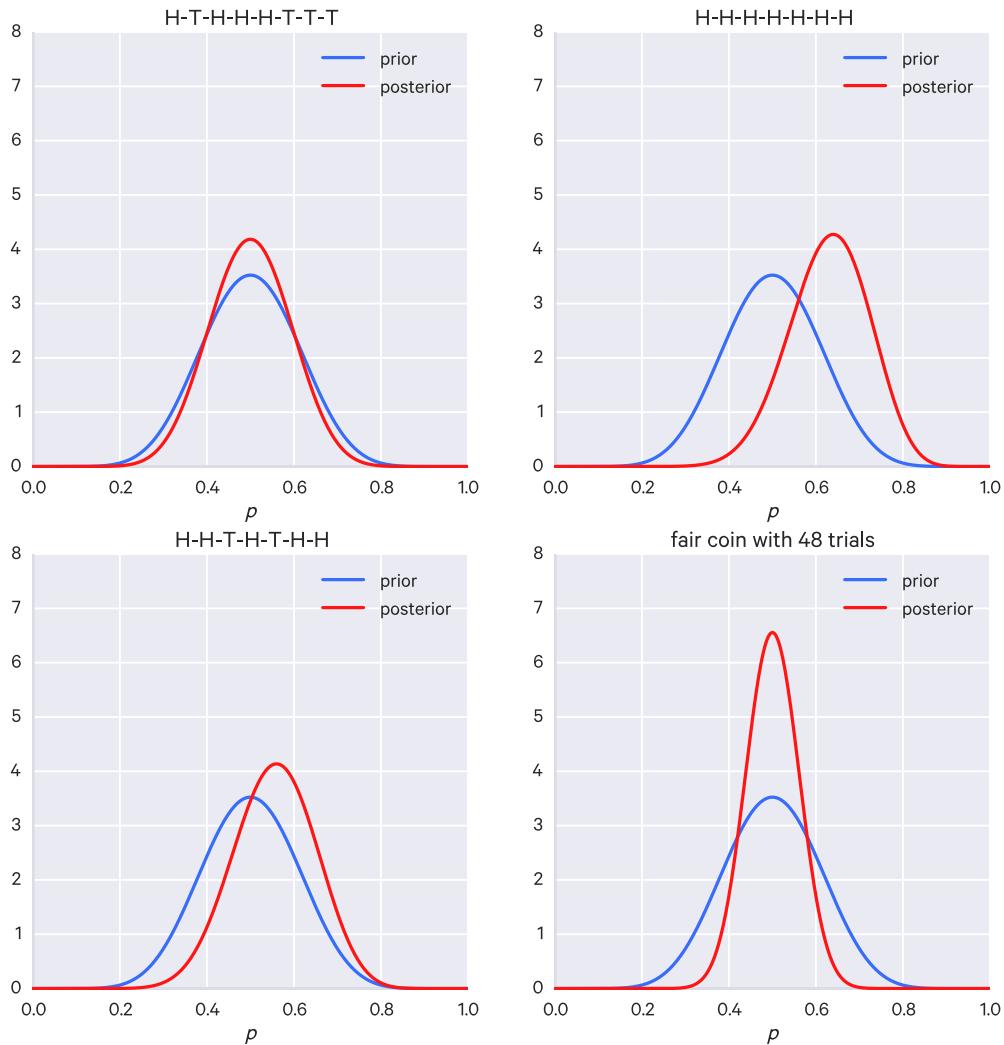
Some possible outcomes with a flat prior

A few things to note here:

- When the evidence has even amounts of tails and heads, the posterior centers around $p = 0.5$.
- When the evidence has even one tail, the possibility of $p = 1$ drops to nothing.
- When the evidence has no tails, the posterior places more weight on an unfair coin, but there is still some possibility of $p = 0.5$. As the number of evidence increases, however, and still no tails show up, the posterior will have even more weight pushed towards $p = 1$.
- When there is a lot of evidence containing even amounts of tails and heads, there is greater confidence that $p = 0.5$ (that is, there's smaller variance around it).

What if instead of a flat prior, we had assumed that the coin was fair to begin with? In this scenario, the α and β values function like counts for heads and tails. So to assume a fair coin we could say,

$\alpha = 10, \beta = 10$. If we have a really strong belief that it is a fair coin, we could say $\alpha = 100, \beta = 100$. The higher these values are, the stronger our belief.



Some possible outcomes with a informative prior

Since our prior belief is stronger than it was with a flat prior, the same amount of evidence doesn't change the prior belief as much. For instance, now if we see a streak of heads, we are less convinced it is unfair.

In either case, we could take the expected value of p 's posterior distribution as our estimate for p , and then use that as evidence for a fair or unfair coin.

6.7.1 References

- POLS 506: Simple Bayesian Models. Justin Esarey. <https://www.youtube.com/watch?v=ps5MYi81lsE>

- POLS 506: Basic Monte Carlo Procedures and Sampling. Justin Esarey. <https://www.youtube.com/watch?v=cxWzsCoYT8Q>
- POLS 506: Metropolis-Hastings, the Gibbs Sampler, and MCMC. Justin Esarey. <https://www.youtube.com/watch?v=j4nEAqUUnVw>
- <http://www.stat.tamu.edu/~fliang/STAT605/lect01.pdf>
- <https://www.youtube.com/watch?v=12eZWG0Z5gY>
- <http://jeremykun.com/2015/04/06/markov-chain-monte-carlo-without-all-the-bullshit/>
- <http://homepages.dcc.ufmg.br/~assuncao/pgm/aulas2014/mcmc-gibbs-intro.pdf>
- <https://plot.ly/ipython-notebooks/computational-bayesian-analysis/>
- *Think Bayes*, Allen Downey.
- Computational Statistics II. Chris Fonnesbeck. SciPy 2015: <https://www.youtube.com/watch?v=heFaYLKVZY4> and https://github.com/fonnesbeck/scipy2015_tutorial
- Bayesian Statistical Analysis. Chris Fonnesbeck. SciPy 2014: https://github.com/fonnesbeck/scipy2014_tutorial
- *Probabilistic Programming and Bayesian Methods for Hackers*, Cam Davidson Pilon: <https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers>
- <https://jakevdp.github.io/blog/2015/08/07/frequentism-and-bayesianism-5-model-selection/>

7

Probabilistic Graphical Models

The key tool for probabilistic inference is the *joint probability table*. Each row in a joint probability table describes a combination of values for a set of random variables. That is, say you have n events which have a binary outcome (T/F). A row would describe a unique configuration of these events, e.g. if $n = 4$ then one row might be 0, 0, 0, 0 and another might be 1, 0, 0, 0 and so on.

Using a joint probability table you can learn a lot about how those events are related probabilistically.

The problem is, however, that joint probability tables can get very big. There will be 2^n rows (for this binary case; with more outcomes naturally leading to even bigger tables), so if you are looking at 10 events you already have 1,024 rows to consider.

We can use **probabilistic graphical models** to reduce this space. Probabilistic graphical models allow us to represent complex networks of interrelated and independent events.

There are two main types of graphical models:

- **Bayesian models:** aka **Bayesian networks**, sometimes called *Bayes nets* or **belief networks**. These are used when there are causal relationships between the random variables.
- **Markov models:** used when there are noncausal relationships between the random variables.

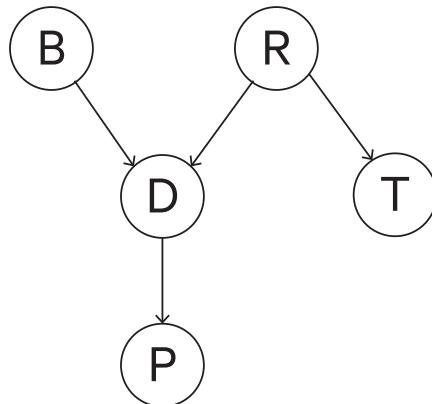
7.1 Bayesian Networks

Say we are looking at five events:

- a dog barking (D)
- a raccoon being present (R)

- a burglar being present (B)
- a trash can is heard knocked over (T)
- the police are called (P)

We can encode some assumptions about how these events are related in a *belief net* (also called a *Bayesian net*):



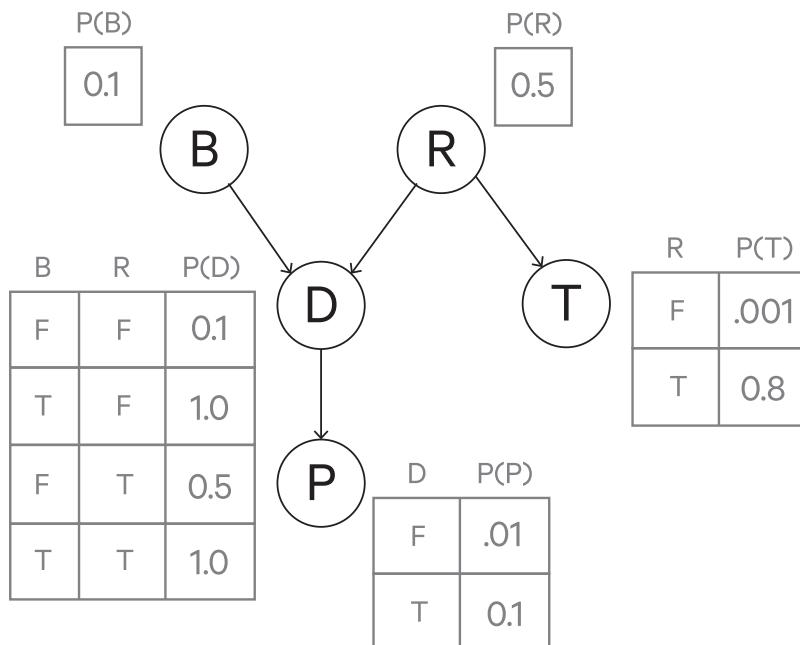
Belief Network

Every node is dependent on its parent and nothing else that is not a descendant. To put it another way: given its parent, a node is independent of all its non-descendants.

For instance, the event P is dependent on its parent D but not B or R or T because their causality flows through D .

D depends on B and R because they are its parents, but not T because it is not a descendant or a parent. But D may depend on P because it is a descendant.

We can then annotate the graph with probabilities:



Belief Network

The B and R nodes have no parents so they have singular probabilities.

The others depend on the outcome of their parents.

With the belief net, we only needed to specify 10 probabilities.

If we had just constructed joint probability table, we would have had to specify $2^5 = 32$ probabilities (rows).

If we expand out the conditional probability of this system using the chain rule, it would look like:

$$P(p, d, b, t, r) = P(p|d, b, t, r)P(d|b, t, r)P(b|t, r)P(t|r)P(r)$$

But we can bring in our belief net's conditional independence assumptions to simplify this:

$$P(p, d, b, t, r) = P(p|d)P(d|b, r)P(b)P(t|r)P(r)$$

Belief networks are *acyclic*, that is, they cannot have any loops (a node cannot have a path back to itself).

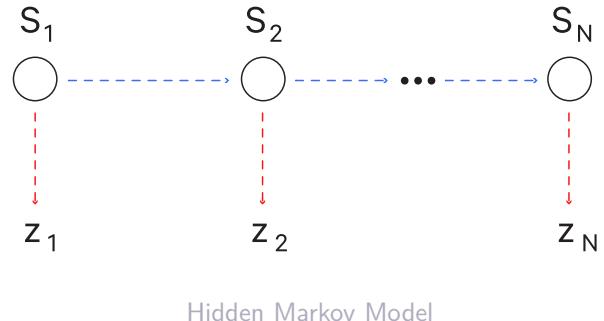
Two nodes (variables) in a Bayes net are on an **active trail** if a change in one node affects the other. This includes cases where the two nodes have a causal relationship, an evidential relationship, or have some common cause.

7.2 Hidden Markov Models (HMM)

Hidden Markov Models are Bayes nets in which each state S_i depends only on the previous state S_{i-1} (so the sequence of states is a Markov chain) and emits an observation z_i . It is hidden because we do not observe the states directly, but only these observations. The actual observations are stochastic (e.g. an underlying state may produce one of many observations with some probability). We try to infer the state based on these observations.

The parameters of a HMM are:

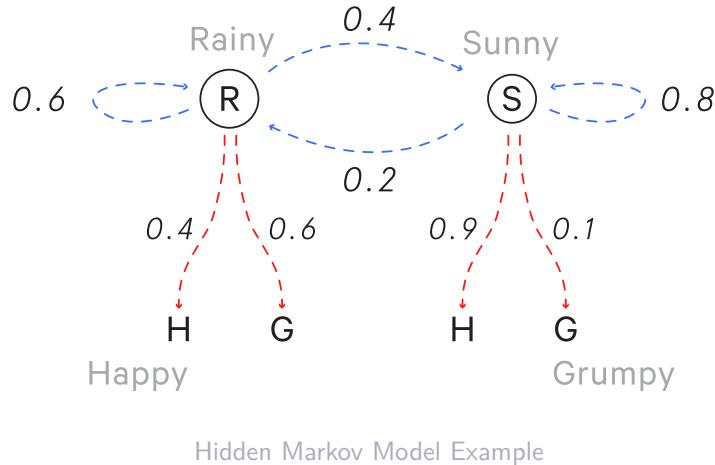
- The initial state distribution, $P(S_0)$
- The transition distribution, $P(S_2|S_1)$
- The measurement/observation distribution, $P(z_1|S_1)$



Hidden Markov Model

7.2.1 Example

Say we have the following HMM:



Hidden Markov Model Example

We don't know the starting state, but we know the probabilities:

$$P(R_0) = \frac{1}{2}$$

$$P(S_0) = \frac{1}{2}$$

Say on the first day we see that this person is happy and we want to know whether or not it is raining. That is:

$$P(R_1|H_1)$$

We can use Bayes' rule to compute this posterior:

$$P(R_1|H_1) = \frac{P(H_1|R_1)P(R_1)}{P(H_1)}$$

We can compute these values by hand:

$$P(R_1) = P(R_1|R_0)P(R_0) + P(R_1|S_0)P(S_0)$$

$$P(H_1) = P(H_1|R_1)P(R_1) + P(H_1|S_1)P(S_1)$$

$P(H_1|R_1)$ can be pulled directly from the graph.

Then you can just run the numbers.

7.3 References

- MIT 6.034 (Fall 2010): Artificial Intelligence. Patrick H. Winston.

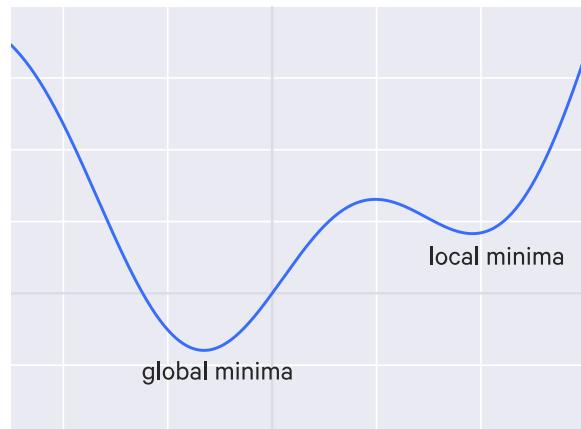
8

Optimization

Optimization is the task of finding the arguments to a function which yield its minimum or maximum value.

Optimization problems can be thought of a topology where you are looking for the global peak (if you are maximizing) or the globally lowest point (if you are minimizing). For simplicity, minimizing will be the assumed goal here, as you are often trying to minimize some error function.

Consider a very naive approach: a greedy random algorithm which starts at some position in this topology, then randomly tries moving to a new position and checks if it is better. If it is, it sets that as the current solution. It continues until it has some reason to stop, usually because it has found a minimum. This is a **local** minimum; that is, it is a minimum relative to its immediately surrounding points, but it is not necessarily the **global** minimum, which is the minimum of the entire function.



Local vs global minima

This algorithm is *greedy* in that it will always prefer a better scoring position, even if it is only marginally better. Thus it can be easy to get stuck in local optima - since any step away from it seems worse, even if the global optimum is right around the corner, so to speak.

8.1 Gradient Descent

Gradient descent (GD) is perhaps the common minimizing optimization (for maximizing, its equivalent is *gradient ascent*) in machine learning.

Say we have a function $C(v)$ which we want to minimize. For simplicity, we will use $v \in \mathbb{R}^2$. An example $C(v)$ is visualized in the accompanying figure.

In this example, the global minimum is visually obvious, but most of the time it is not (especially when dealing with far more dimensions). But we can apply the model of a ball rolling down a hill and expand it to any arbitrary n dimensions. The ball will “roll” down to a minimum, though not necessarily the global minimum.

The position the ball is at is a potential solution; here it is some values for v_1 and v_2 . We want to move the ball such that ΔC , the change in $C(v)$ from the ball’s previous position to the new position, is negative (i.e. the cost function’s output is smaller, since we’re minimizing).

More formally, ΔC is defined as:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2$$

We define the *gradient* of C , denoted ∇C , to be the vector of partial derivatives (transposed to be a column vector):

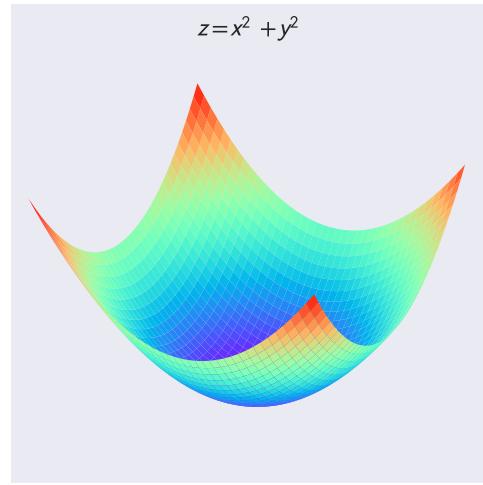
$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T$$

So we can rewrite ΔC as:

$$\Delta C \approx \nabla C \cdot \Delta v.$$

We can choose Δv to make ΔC negative:

$$\Delta v = -\eta \nabla C,$$



Sphere function

Where η is a small, positive parameter (the **learning rate**), which controls the step size.

Finally we have:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$$

We can use this to compute a value for Δv , which is really the change in position for our “ball” to a new position v' :

$$v \rightarrow v' = v - \eta \nabla C.$$

And repeat until we hit a global (or local) minimum.

This process is known in particular as **batch gradient descent** because each step is computed over the entire *batch* of data.

8.1.1 Stochastic gradient descent (SGD)

With batch gradient descent, the cost function is evaluated on all the training inputs for each step. This can be quite slow.

With **stochastic gradient descent** (SGD), you randomly shuffle your examples and look at only one example for each iteration of gradient descent. Ultimately it is less direct than batch gradient descent but gets you close to the global minimum - the main advantage is that you’re not iterating over your entire training set for each step, so though its path is more wandering, it ends up taking less time on large datasets.

In fact, stochastic gradient descent can help with finding the global minimum because instead of computing over a single error surface, you are working with many different error surfaces varying with the example you are current looking at. So it is possible that in one of these surfaces a local minima does not exist or is less pronounced than in others, which make it easier to surpass.

There’s another form of stochastic gradient descent called **minibatch gradient descent**. Here b random examples are used for each iteration, where b is your minibatch size. It is usually in the range of 2-100; a typical choice might be 10. Note that a minibatch size can be *too* large, resulting in greater time for convergence. But generally it is faster than SGD and has the benefit of aiding in local minima avoidance.

When the stochastic variant is used, a $\frac{1}{b}$ term is sometimes included:

$$v \rightarrow v' = v - \frac{\eta}{b} \nabla C.$$

8.1.2 Learning rates

The learning rate η is typically held constant. It can be slowly decreased it over time if you want θ to converge on the global minimum in stochastic gradient descent (otherwise, it just gets close). So for instance, you can divide it by the iteration number plus some constant, but this can be overkill.

8.2 Simulated Annealing

Simulated annealing is similar to the greedy random approach but it has some randomness which can “shake” it out of local optima.

Annealing is a process in metal working where the metal starts at a very high temperature and gradually cools down. Simulated annealing uses a similar process to manage its randomness.

A simulated annealing algorithm starts with a high “*temperature*” (or “*energy*”) which “cools” down (becomes less extreme) as progress is made. Like the greedy random approach, the algorithm tries a random move. If the move is better, it is accepted as the new position. If the move is worse, then there is a chance it still may be accepted; the probability of this is based on the current temperature, the current error, and the previous error:

$$P(e, e', T) = \exp\left(\frac{-(e' - e)}{T}\right)$$

Each random move, whether accepted or not, is considered an iteration. After each iteration, the temperature is decreased according to a *cooling schedule*. An example cooling schedule is:

$$T(k) = T_{\text{init}} \frac{T_{\text{final}}}{T_{\text{init}}}^{\frac{k}{k_{\text{max}}}}$$

where

- T_{init} = the starting temperature
- T_{final} = the minimum/ending temperature
- k = the current iteration
- k_{max} = the maximum number of iterations

For this particular schedule, you probably don’t want to set T_{final} to 0 since otherwise it would rapidly decrease to 0. Set it something close to 0 instead.

The algorithm terminates when the temperature is at its minimum.

8.3 Nelder-Mead (aka Simplex or Amoeba optimization)

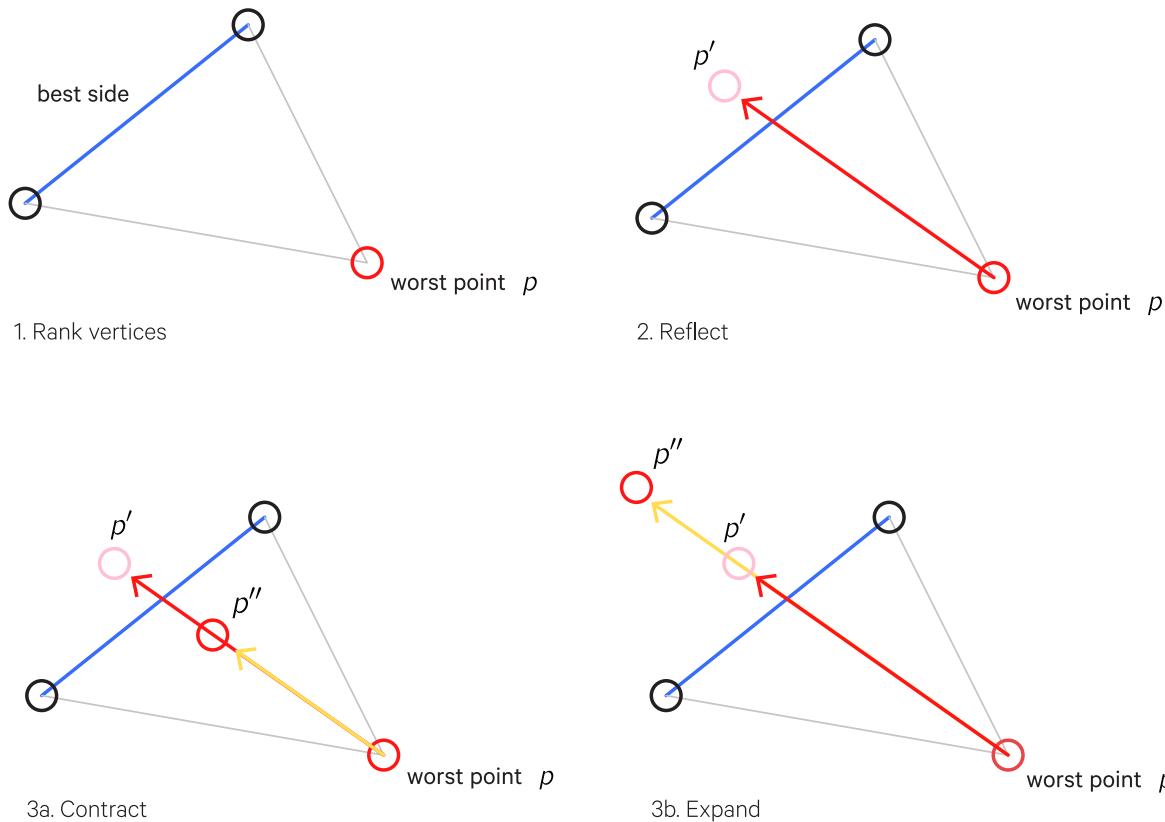
For a problem of n dimensions, create a shape of $n + 1$ vertices. This shape is a **simplex**.

One vertex of the simplex is initialized with your best educated guess of the solution vector. That guess could be the output of some other optimization approach, even a previous Nelder-Mead run. If you have nothing to start with, a random vector can be used.

The other n vertices are created by moving in one of the n dimensions by some set amount.

Then at each step of the algorithm, you want to (illustrations are for $n = 2$, thus 3 vertices):

- Find the worst, second worst, and best scoring vertices
- Reflect the worst vertex to some point p' through the best side
- If p' is better, *expand* by setting the worst vertex to a new point p'' , a bit further than p' but in the same direction
- If p' is worse, then *contract* by setting the worst vertex to a new point p'' , in the same direction as p' but before crossing the best side



The algorithm terminates when one of the following occurs:

- The maximum number of iterations is reached
- The score is “good enough”
- The vertices have become close enough together

Then the best vertex is considered the solution.

This optimization method is very sensitive to how it is initialized; whether or not a good solution is found depends a great deal on its starting points.

8.4 Particle Swarm Optimization

Particle swarm optimization is similar to Nelder-Mead, but instead of three points, many more points are used. These points are called “particles”.

Each particle has a position (a potential solution) and a velocity which indicates where the particle moves to in the next iteration. Particles also keep track of their current error to the training examples and its best position so far.

Globally, we also track the best position overall and the lowest error overall.

The velocity for each particle is computed according to:

- it's inertia (i.e. the current direction it is moving in)
- it's historic best position (i.e. the best position it's found so far)
- the global best position

The influence of these components are:

- inertia weight
- cognitive weight (for historic best position)
- social weight (for global best position)

These weights are parameters that must be tuned, but this method is quite robust to them (that is, they are not sensitive to these changes so you don't have to worry too much about getting them just right).

More particles are better, of course, but more intensive.

You can specify the number of epochs (iterations) to run.

You can also incorporate a death-birth cycle in which low-performing particles (those that seem to be stuck, for instance) get destroyed and a new randomly-placed particle is initialized in its place.

8.5 Genetic Algorithms

- You have a *population* of “chromosomes” (e.g. possible solutions or parameters)
- There may be some *mutation* in the chromosomes (e.g. with binary chromosomes, sometimes 0s become 1s and vice versa or with continuous values, changes happen according to some step size)
- Some chromosomes *crossover* where the front part of one chromosome combines with the back part of another
- The *genotype* (the chromosome composition) is expressed as some *phenotype* (i.e. some genetically-determined properties) in some individuals
- Then each of these individuals has some *fitness* value resulting from their phenotypes
- These fitnesses are turned into some probability of *survival*
- Then the individuals are *selected* randomly based on their survival probabilities
- These individuals form the new chromosome population for the next *generation*

Each of these steps requires some decisions by the implementer.

For instance, how do you translate a fitness score into a survival probability?

Well, the simplest way is:

$$P_i = \frac{f_i}{\sum_i f_i}$$

Where f_i is the fitness of some individual i .

However, depending on how you calculate fitness, this may not be appropriate.

You could alternatively use a ranking method, in which you just look at the relative fitness rankings and not their actual values. So the most fit individual is most likely to survive, the second fit is a bit less likely, and so on.

You pick a probability constant P_C , and the survival of the top-ranked individual is P_C , that of the second is $(1 - P_C)P_C$, that of the third is, $(1 - P_C)^2P_C$, and so on. So $P_{n-1} = (1 - P_C)^{n-2}P_C$ and $P_n = (1 - P_C)^{n-1}$.

If you get stuck on local maxima you can try increasing the step size. When your populations start to get close to the desired value, you can decrease the step size so the changes are less sporadic (i.e. use simulated annealing).

When selecting a new population, you can incorporate a diversity rank in addition to a fitness rank. This diversity ranking tries to maximize the diversity of the new population. You select one individual for the new population, and then as you select your next individual, you try and find one which is distinct from the already selected individuals.

8.6 Derivative-Free Optimization

Note that Nelder-Mead, Particle Swarm, and genetic algorithm optimization methods are sometimes known as “derivative-free” because they do not involve computing derivatives in order to optimize.

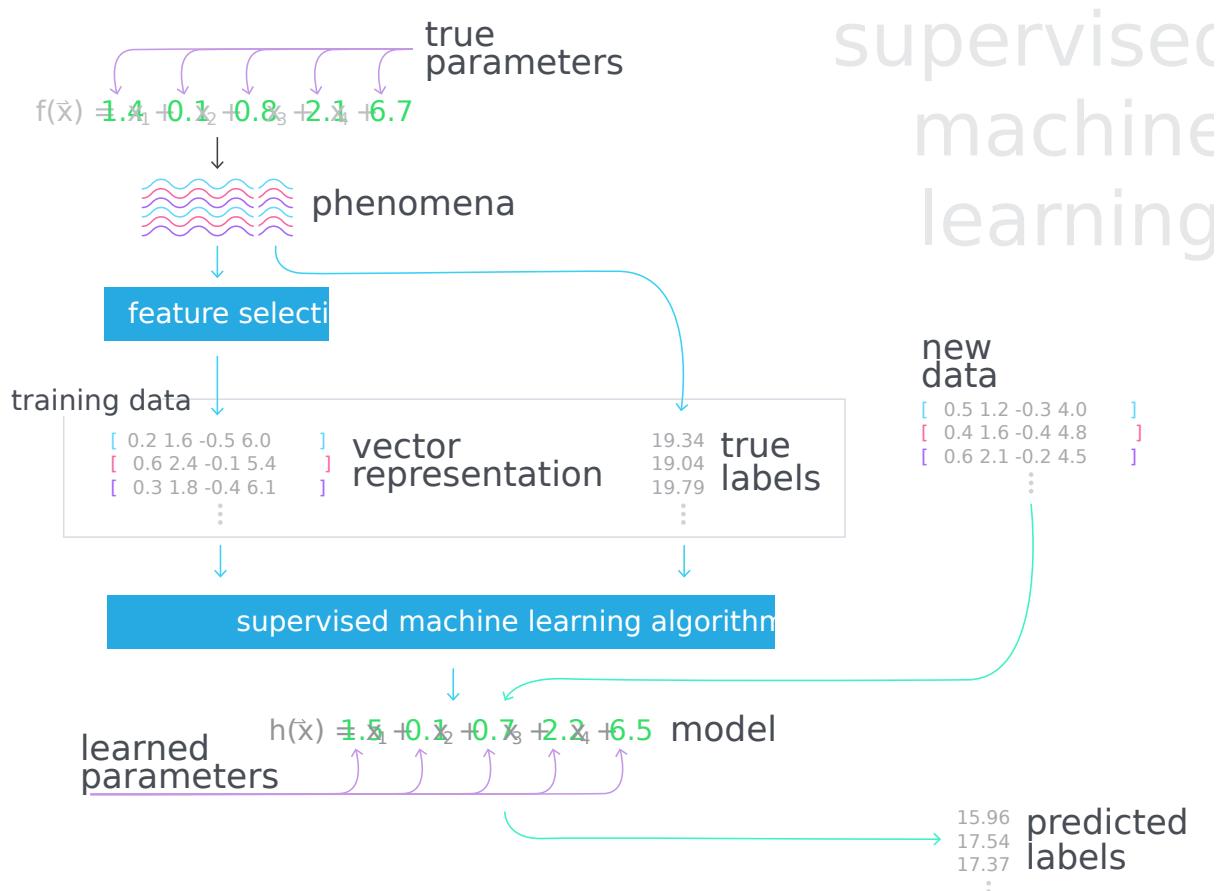
8.7 References

- Swarm Intelligence Optimization using Python (PyData 2015). James McCaffrey. https://www.youtube.com/watch?v=bVDX_UwthZI
- Andrew Ng's Coursera *Machine Learning* course (2014)

Part II

Machine Learning

Machine Learning



The general process of (supervised) machine learning

9.1 Representation vs Learning:

- Representation: whether or not a function can be simulated by the network; i.e. is the network capable of *representing* a given function?
- Learning: whether or not there exists an algorithm with which the weights can be adjusted to represent a particular function

9.2 References

- *Neural Computing: Theory and Practice* (1989). Philip D. Wasserman

9.3 Supervised Learning

The learning algorithm is provided some pre-labeled examples (a *training set*) to learn from.

In *regression* problems, you try to predict some continuous valued output (i.e. a real number).

In *classification* problems, you try to predict some discrete valued output (e.g. categories).

Typical notation:

- m = number training of examples
- x 's = input variables or features
- y 's = output variables or the “target” variable
- $(x^{(i)}, y^{(i)})$ = the i th training example
- h = the hypothesis, that is, the function that the learning algorithm learns, taking x 's as input and outputting y 's

The typical process is:

- Feed training set data into the learning algorithm
- The learning algorithm learns the hypothesis h
- Input new data into h
- Get output from h

The hypothesis can thought of as the model that you try to learn for a particular task. You then use this model on new inputs, e.g. to make predictions - **generalization** is how the model performs on new examples; this is most important in machine learning.

9.4 Unsupervised Learning

The learning algorithm is not provided with any pre-labeled examples. Generally you are trying to uncover some structure or patterns in the data.

An example is a *clustering* algorithm. We don't tell the algorithm in advance anything about the structure of the data; it discovers it on its own by figuring how to group them.

Some other examples are *dimensionality reduction*, in which you try to reduce the dimensionality of the data representation, *density estimation*, in which you estimate the density of the distribution of the data, and *feature extraction*, in which you try to learn meaningful features automatically.

9.5 Other types of learning

9.5.1 Semi-supervised Learning

Some labeled data, some unlabeled data.

9.5.2 Active Learning

Like semi-supervised learning, you have some labeled data, and some unlabeled data, and you can ask for the labels of some of the unlabeled data.

9.5.3 Reinforcement Learning

Actions are taken and rewarded or penalized in some way and the goal is maximizing lifetime/long-term reward (or minimizing lifetime/long-term penalty).

9.6 Representation

A very important choice in machine learning is how you represent the data. What are its salient features, and in what form is it best presented? Each field in the data (e.g. column in the table) is a **feature** and a great deal of time is spent getting this representation right. The best machine learning algorithms can't do much if the data isn't represented in a way suited to the task at hand.

Sometimes it's not clear how to represent data. For instance, in identifying an image of a car, you may want to use a wheel as a feature. But how do you define a wheel in terms of pixel values?

Representation learning is a kind of machine learning in which representations themselves can be learned.

An example representation learning algorithm is the **autoencoder**. It's a combination of an *encoder* function that converts input data into a different representation and a *decoder* function which converts the new representation back into its original format.

Successful representations separate the *factors of variations* (that is, the contributors to variability) in the observed data. These may not be explicit in the data, “they may exist either as unobserved objects or forces in the physical world that affect the observable quantities, or they are constructs in the human mind that provide useful simplifying explanations or inferred causes of the observed data.” ([Deep Learning](#)).

9.6.1 Deep Learning

Deep learning builds upon representation learning. It involves having the program learn some hierarchy of concepts, such that simpler concepts are used to construct more complicated ones. This hierarchy of concepts forms a deep (many-layered) graph, hence “deep learning”.

With deep learning we can have simpler representations aggregate into more complex abstractions.

A basic example of a deep learning model is the multilayer perceptron (MLP), which is essentially a function composed of simpler functions (layers); each function (i.e. layer) can be thought of as taking the input and outputting a new representation of it.

For example, if we trained a MLP for image recognition, the first layer may end up learning representations of edges, the next may see corners and contours, the next may identify higher level features like faces, etc.

9.6.2 References

- [Deep Learning](#). Yoshua Bengio, Ian J. Goodfellow, Aaron Courville. Book in preparation for MIT Press. 2015.

9.7 Terminology

- **Capacity**: the flexibility of a model
- **Hyperparameter**: a parameter of a model that is not learned (that is, you specify it yourself)
- **Underfitting**: when the model could achieve better generalization with more training or capacity
- **Overfitting**: when the model could achieve better generalization with more training or capacity; in particular, the model is too tuned to the idiosyncrasies of the training data
- **Model selection**: the process of choosing the best hyperparameters on a validation set

9.8 Optimization

Much of machine learning can be framed as optimization problems - there is some kind of objective or loss function which we want to optimize (e.g. minimize classification error on the training set). Typically you are trying to find some parameters for your model, θ , which minimizes this objective or loss function.

Generally this framework for machine learning is called **empirical risk minimization** and can be formulated:

$$\operatorname{argmin}_{\theta} \frac{1}{n} \sum_i l(f(x^{(i)}; \theta), y^{(i)}) + \lambda \Omega(\theta)$$

Where:

- $f(x^{(i)}; \theta)$ is your model, which outputs some predicted value for the input $x^{(i)}$ and θ are the parameters for the model
- $y^{(i)}$ is the training label (i.e. the ground-truth) for the input $x^{(i)}$
- l is the loss function
- $\Omega(\theta)$ is a *regularizer* to penalize certain values of θ and λ is the regularization parameter (see below on *regularization*)

Some optimization terminology:

- **Critical points:**
 $x \in \mathbb{R}^n | \nabla_x f(x) = 0$
- **Curvature in direction** $v: v^T \nabla_x^2 f(x) v$
- Types of critical points:
 - local minima: $v^T \nabla_x^2 f(x) v > 0, \forall v$, that is $\nabla_x^2 f(x)$ is positive definite
 - local maxima: $v^T \nabla_x^2 f(x) v < 0, \forall v$, that is $\nabla_x^2 f(x)$ is negative definite
 - saddle point: curvature is positive in some directions and negative in others

9.9 Linear Regression with One Variable

Also known as *univariate linear regression* or *simple linear regression* (SLR).

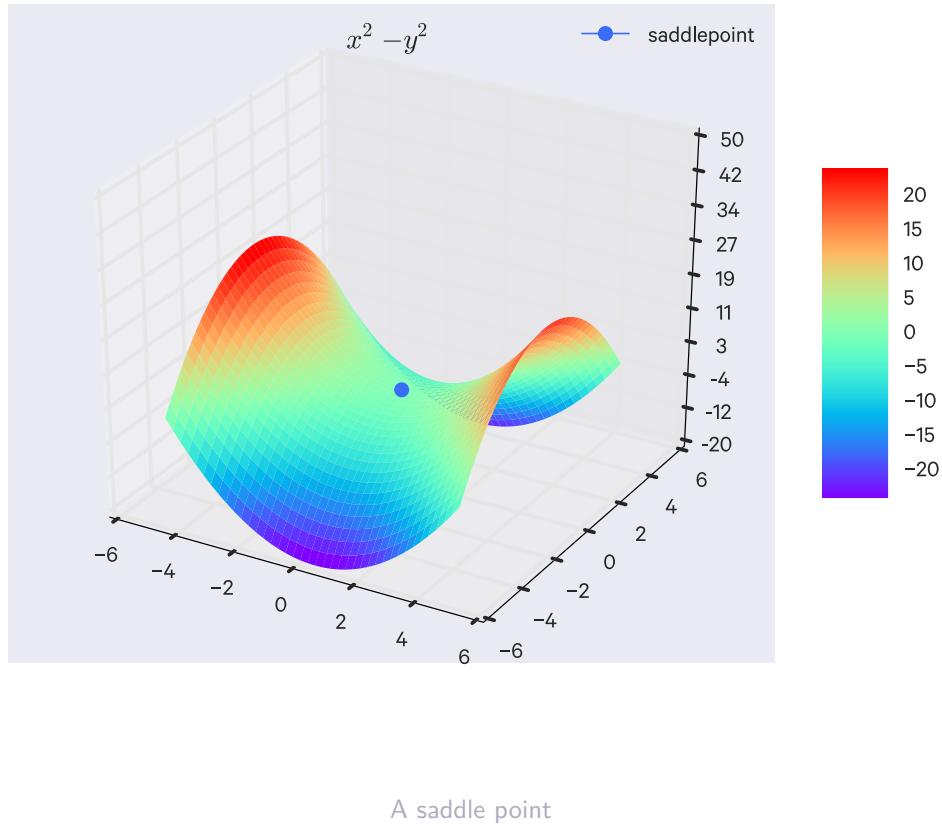
In univariate linear regression, we have one input variable x .

The hypothesis takes the form:

$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

Where the θ 's are the *parameters* that the learning algorithm learns.

This should look familiar: it's just a line.



9.9.1 How are the parameters determined?

The general idea is that you want to choose your parameters so that $h_\theta(x)$ is close to y for your training examples (x, y) . This can be written:

$$\sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

To make the math easier, you multiply everything by $\frac{1}{2m}$ (this won't affect the resulting parameters):

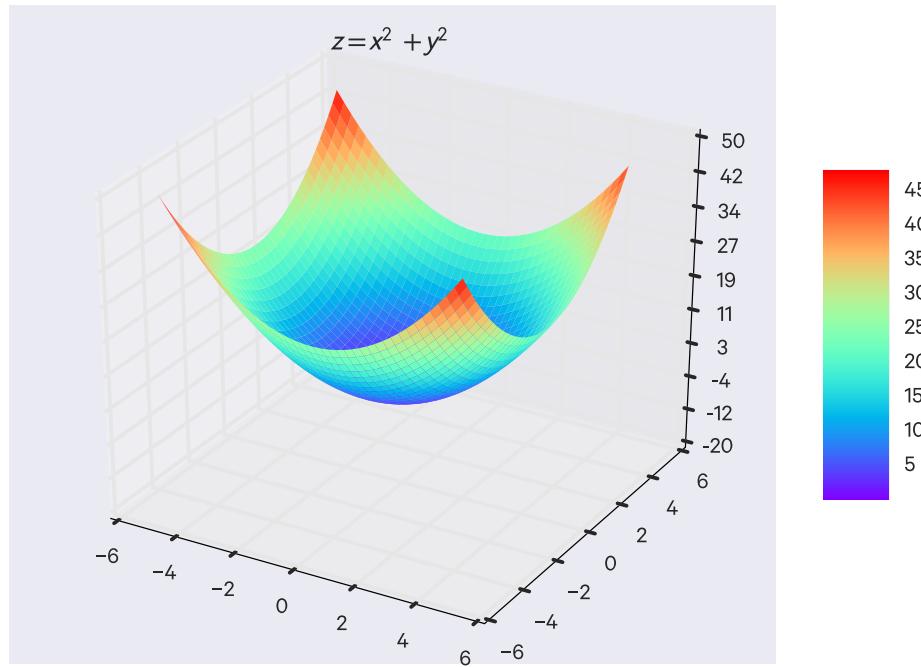
$$\frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

This is the *cost function* (or *objective function*). In this case, we call it J , which looks like:

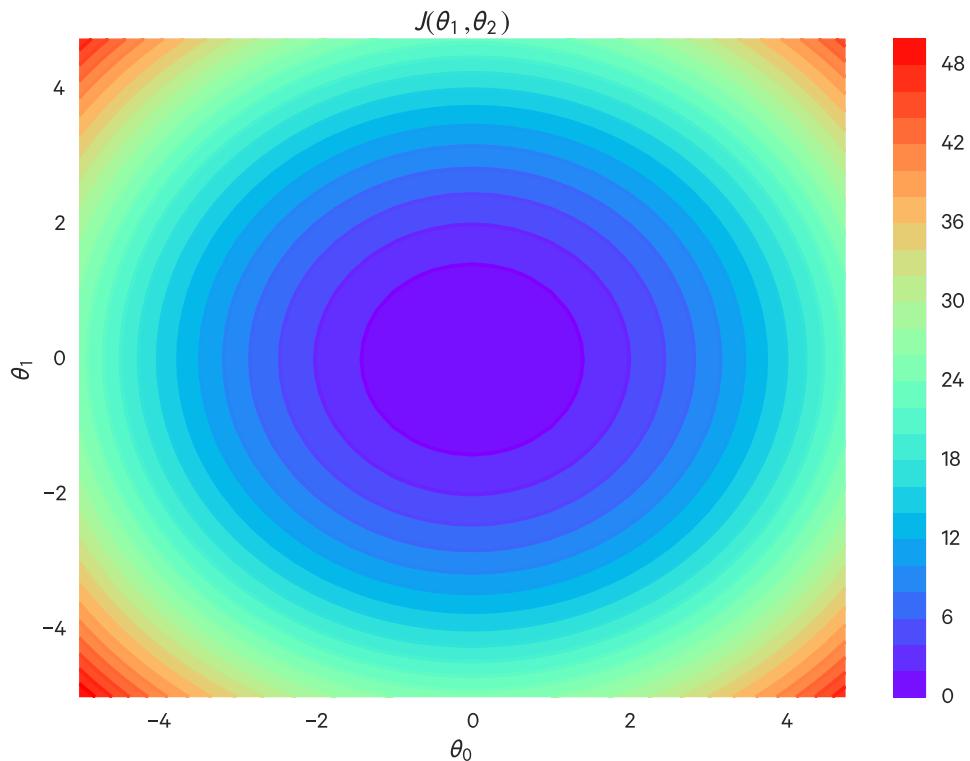
$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

Here it is the *squared error function* - it is probably the most commonly used cost function for regression problems.

We want to find (θ_0, θ_1) to *minimize* $J(\theta_0, \theta_1)$.



An example cost function with two parameters



The same cost function, visualized as a contour plot

9.10 Gradient Descent

Gradient descent is an algorithm for finding parameter values which minimize a cost function.

Gradient descent is used in a lot of optimization problems across machine learning.

So we have some cost function $J(\theta_0, \theta_1, \dots, \theta_n)$ and we want to minimize it.

The general approach is:

- Start with some $\theta_0, \theta_1, \dots, \theta_n$.
- Changing $\theta_0, \theta_1, \dots, \theta_n$ in some increment/step to reduce $J(\theta_0, \theta_1, \dots, \theta_n)$ as much as possible.
- Repeat the previous step until convergence on a minimum (hopefully)

Gradient descent algorithm

Repeat the following until convergence: (Note that $:=$ is the assignment operator.)

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1, \dots, \theta_n)$$

For each j in n .

Every θ_j is updated *simultaneously*. So technically, you'd calculate this value for each j in n and only after they are all updated would you actually update each θ_j .

For example, if the right-hand side of that equation was a function `func(j, t0, t1)`, you would implement it like so (example is $n = 2$):

```
temp_0 = func(0, theta_0, theta_1)
temp_1 = func(1, theta_0, theta_1)
theta_0 = temp_0
theta_1 = temp_1
```

α is the *learning rate* and tells how large a step/increment to change the parameters by.

Learning rates which are too small cause the gradient descent to go slowly. Learning rates which are too large can cause the gradient descent to overshoot the minimum, and in those cases it can fail to converge or even diverge.

The partial derivative on the right is just the rate of change from the current value.

9.10.1 Gradient Descent for Univariate Linear Regression

For univariate linear regression, the derivatives are:

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})$$

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

so overall, the algorithm involves repeatedly updating:

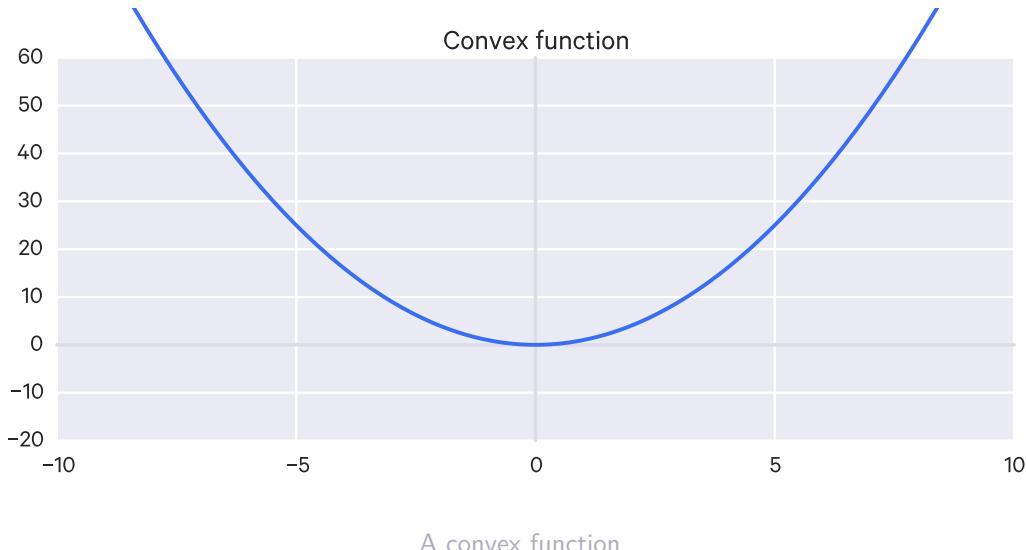
$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

Remember that the θ parameters are updated *simultaneously*.

Note that because we are summing over *all* training examples for each step, this particular type of gradient descent is known as *batch gradient descent*. There are other approaches which only sum over a subset of the training examples for each step.

Univariate linear regression's cost function is always convex ("bowl-shaped"), which has only one optimum, so gradient descent in this case will always find the global optimum.



9.11 Linear Regression with Multiple Variables

Also known as *multivariate* linear regression. This technique is for using multiple features with linear regression.

Say we have:

- n = number of features
- $x^{(i)}$ = the input features of the i th training example

- $x_j^{(i)}$ = the value of feature j in the i th training example

Instead of the simple linear regression model we can use a **generalized linear model** (GLM). That is, the hypothesis h will take the form of:

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

For convenience of notation, you can define $x_0 = 1$ and notate your features and parameters as zero-indexed $n+1$ -dimensional vectors:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

And the hypothesis can be re-written as:

$$h_{\theta}x = \theta^T x$$

9.12 Gradient descent with Multivariate Linear Regression

The previous gradient descent algorithm for univariate linear regression is just generalized (this is still repeated and simultaneously updated):

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

9.12.1 Feature Scaling

If you design your features such that they are on a similar scale, gradient descent can converge more quickly.

For example, say you are developing a model for predicting the price of a house. Your first feature may be the area, ranging from 0-2000 sqft, and your second feature may be the number of bedrooms, ranging from 1-5.

These two ranges are very disparate, causing the contours of the cost function to be such that the gradient descent algorithm jumps around a lot trying to find an optimum.

If you *scale* these features such that they share the same (or at least a similar) range, you avoid this problem.

More formally, with feature scaling you want to get every feature into approximately a $-1 \leq x_i \leq 1$ range (it doesn't necessarily have to be between -1 and 1, just so long as there is a consistent range across your features).

With feature scaling, you could also apply *mean normalization*, where you replace x_i with $x_i - \mu_i$ (that is, replace the value of the i th feature with its value minus the mean value for that feature) such that the mean of that feature is shifted to be about zero (note that you wouldn't apply this to $x_0 = 1$).

9.12.2 Choosing the Learning Rate α

You can plot out a graph with the number of gradient descent iterations on the x-axis and the values of $\min_{\theta} J(\theta)$ on the y-axis and visualize how the latter changes with the number of iterations. At some point, that curve will flatten out; that's about the number of iterations it took for gradient descent to converge on your particular problem.



You could use an *automatic convergence test* which just declares convergence if $J(\theta)$ decreases by less than some threshold value in an iteration, but in practice that threshold value may be difficult to determine.

You would expect this curve to be similar to the one above. $\min_{\theta} J(\theta)$ should decrease with the number of iterations, if gradient descent is working correctly. If not, then you should probably be using a smaller learning rate (α). But again, don't make it too small or convergence will be slow.

9.13 Example implementation of linear regression with gradient descent

```
"""
- X = feature vectors
- y = labels/target variable
- theta = parameters
- hyp = hypothesis (actually, the vector computed from the hypothesis function)
"""

import numpy as np

def cost_function(X, y, theta):
    """
    This isn't used, but shown for clarity
    """

    m = y.size
    hyp = np.dot(X, theta)
    sq_err = sum(pow(hyp - y, 2))
    return (0.5/m) * sq_err

def gradient_descent(X, y, theta, alpha=0.01, iterations=10000):
    m = y.size
    for i in range(iterations):
        hyp = np.dot(X, theta)
        for i, p in enumerate(theta):
            temp = X[:,i]
            err = (hyp - y) * temp
            cost_function_derivative = (1.0/m) * err.sum()
            theta[i] = theta[i] - alpha * cost_function_derivative
    return theta

if __name__ == '__main__':
    def true_function(X):
        # Create random parameters for X's dimensions, plus one for x0.
        true_theta = np.random.rand(X.shape[1] + 1)
        return true_theta[0] + np.dot(true_theta[1:], X.T), true_theta

    # Create some random data
```

```

n_samples = 20
n_dimensions = 5
X = np.random.rand(n_samples, n_dimensions)
y, true_theta = true_function(X)

# Add a column of 1s for x0
ones = np.ones((n_samples, 1))
X = np.hstack([ones, X])

# Initialize parameters
theta = np.zeros((n_dimensions+1))

# Split data
X_train, y_train = X[:-1], y[:-1]
X_test, y_test = X[-1:], y[-1:]

# Estimate parameters
theta = gradient_descent(X_train, y_train, theta, alpha=0.01, iterations=10000)

# Predict
print('true', y_test)
print('pred', np.dot(X_test, theta))

print('true theta', true_theta)
print('pred theta', theta)

```

9.14 Polynomial Regression

Your data may not fit a straight line and might be better described by a polynomial function, e.g. $\theta_0 + \theta_1x + \theta_2x^2$ or $\theta_0 + \theta_1x + \theta_2x^2 + \theta_3x^3$.

A trick to this is that you can write this in the form of plain old multivariate linear regression. You would, for example, just treat x as a feature x_1 , x^2 as another feature x_2 , x^3 as another feature x_3 , and so on:

$$\theta_0 + \theta_1x_1 + \theta_2x_2 + \theta_3x_3 + \dots + \theta_nx_n$$

Note that in situations like this, feature scaling is very important because these features' ranges differ by a lot due to the exponents.

9.15 Feature Engineering

Your data may have features explicitly present, e.g. a column in a database. But you can also design or *engineer* new features by combining these explicit features or through observing patterns on your own in the data that haven't yet been explicitly encoded. We're doing a form of this in polynomial regression above by encoding the polynomials as new features.

9.16 Normal Equation

The normal equation is an approach which allows for the direct determination of an optimal θ without the need for an iterative approach like gradient descent.

With calculus, you find the optimum of a function by calculating where its derivatives equal 0 (the intuition is that derivatives are rates of change, when the rate of change is zero, the function is “turning around” and is at a peak or valley).

So we can take the same cost function we've been using for linear regression and take the partial derivatives of the cost function J with respect to every parameter of θ and then set each of these partial derivatives to 0:

$$J(\theta_0, \theta_1, \dots, \theta_m) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

And for each j

$$\frac{\partial}{\partial \theta_j} J(\theta) = \dots = 0$$

Then solve for $\theta_0, \theta_1, \dots, \theta_m$.

The fast way to do this is to construct a matrix out of your features, including a column for $x_0 = 1$ (so it ends up being an $m \times (n + 1)$ dimensional matrix) and then construct a vector out of your target variables y (which is an m -dimensional vector):

If you have m examples, $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$, and n features and then include $x_0 = 1$, you have the following feature vectors:

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1}$$

From which we can construct \mathbf{X} , known as the *design matrix*:

$$\mathbf{X} = \begin{bmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(m)})^T \end{bmatrix}$$

That is, the design matrix is composed of the transposes of the feature vectors for all the training examples.

And then the vector y is just all of the labels from your training data:

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Then you can calculate the θ vector which minimizes your cost function like so:

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

With this method, feature scaling isn't necessary.

Note that it's possible that $\mathbf{X}^T \mathbf{X}$ is not invertible (that is, it is *singular*, also called *degenerate*), but this is usually due to redundant features (e.g. having a feature in feet and in meters; they communicate the same information) or having too many features (e.g. $m \leq n$), in which case you should delete some features or use *regularization*.

Programs which calculate the inverse of a matrix often have a method which allows it to calculate the optimal θ vector even if $\mathbf{X}^T \mathbf{X}$ is not invertible.

9.16.1 Deciding between Gradient Descent and the Normal Equation

- Gradient Descent
 - requires that you choose α
 - needs many iterations
 - works well when n is large
- Normal Equation
 - don't need to choose α
 - don't need to iterate
 - slow if n is very large (computing $(\mathbf{X}^T \mathbf{X})^{-1}$ has a complexity of $O(n^3)$), but is usually ok up until around $n = 10000$

Also note that for some learning algorithms, the normal equation is not applicable, whereas gradient descent still works.

9.17 Classification

Classification problems are where your target variables are discrete, so they represent categories or classes.

In cases of *binary classification*, where there are only two classes (that is, $y \in \{0, 1\}$), we call the 0 class the *negative* class, and the 1 class the *positive* class.

9.17.1 Logistic Regression

Logistic regression is a common approach to classification (the name “regression” makes it a bit confusing, it is a classification algorithm).

Logistic regression outputs a value between zero and one (that is, $0 \leq h_\theta(x) \leq 1$).

Say we have our hypothesis function

$$h_\theta(x) = \theta^T x$$

With logistic regression, we apply an additional function g :

$$h_\theta(x) = g(\theta^T x)$$

where

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

This function is known as the *sigmoid* function, also known as the *logistic* function, with the form:

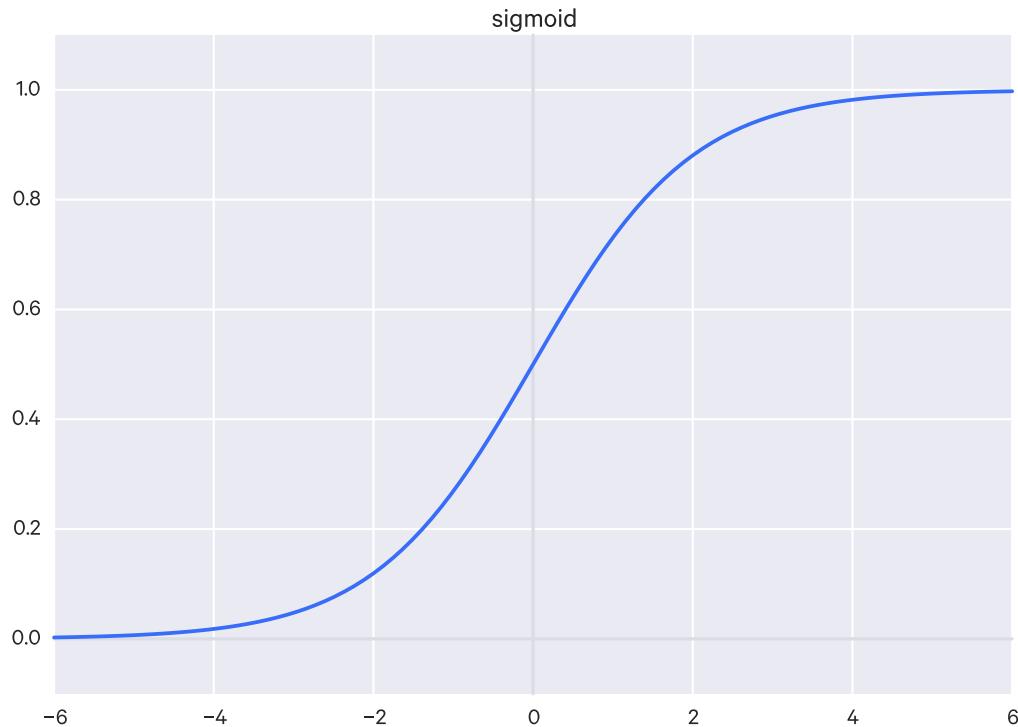
So in logistic regression, the hypothesis ends up being:

$$h_\theta(x) = \frac{1}{1 + e^{\theta^T x}}$$

The output of the hypothesis is interpreted as the probability of the given input belonging to the positive class. that is:

$$h_\theta(x) = P(y = 1|x; \theta)$$

Which is read: “the probability that $y = 1$ given x as parameterized by θ ”.



The sigmoid or logistic function

Since we are classifying input, we want to output a label, not a continuous value. So we might say $y = 1$ if $h_\theta(x) \geq 0.5$ and $y = 0$ if $h_\theta(x) < 0.5$. The line that forms this divide is an example of a *decision boundary*. Note that decision boundaries can be non-linear as well (e.g. they could be a circle or something).

Cost function

So we have our training set $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$ where $y \in \{0, 1\}$ and with the hypothesis function from before.

Here is the cost function for *linear* regression:

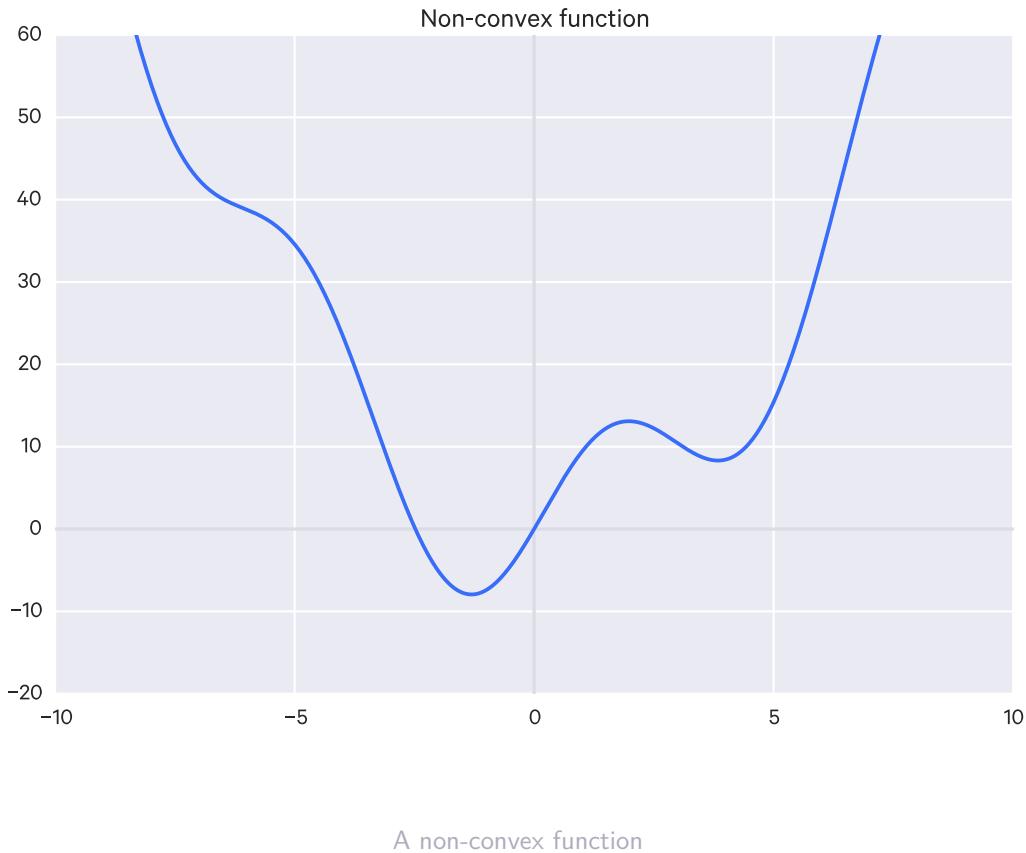
$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Note that the $\frac{1}{2}$ is introduced for convenience, so that the square exponent cancels out when we differentiate. Introducing an extra constant doesn't affect the result.

Note that now the $\frac{1}{2m}$ has been split into $\frac{1}{m}$ and $\frac{1}{2}$.

We can extract $\frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$ and call it $\text{Cost}(h_\theta(x), y)$.

The cost function for logistic regression is different than that used for linear regression because the hypothesis function of logistic regression causes $J(\theta)$ to be *non-convex*, that is, look something like the following with many local optima, making it hard to converge on the global minimum.



So we want to find a way to define $\text{Cost}(h_\theta(x), y)$ such that it gives us a convex $J(\theta)$. We will use:

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

Some properties of this function is that if $y = h_\theta(x)$ then $\text{Cost} = 0$, and as $h_\theta(x) \rightarrow 0$, $\text{Cost} \rightarrow \infty$.

We can rewrite cost in a form more conducive to gradient descent:

$$\text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))$$

So our entire cost function is:

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]$$

You could use other cost functions for logistic regression, but this one is derived from the principle of maximum likelihood estimation and has the nice property of being convex, so this is the one that basically everyone uses for logistic regression.

Then we can calculate $\min_{\theta} J(\theta)$ with gradient descent by repeating and simultaneously updating:

$$\theta_j := \theta_j - \alpha \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

This looks exactly the same as the linear regression gradient descent algorithm, but it is different because $h_\theta(x)$ is now the nonlinear $h_\theta(x) = \frac{1}{1+e^{\theta^T x}}$. Still, the previous methods for gradient descent (feature scaling and learning rate adjustment) apply here.

Advanced optimization algorithms

A few more advanced optimization algorithms are available beyond just gradient descent:

- Conjugate gradient
- BFGS
- L-BFGS

These shouldn't be implemented on your own since they require a *very* advanced understanding of numerical computing, even just to understand what they're doing.

They are more complex, but there's no need to manually pick α and they are often faster than gradient descent. So you can take advantage of them via some library which has them implemented (though some implementations are better than others).

Logistic regression and multiclass classification (One-vs-All)

Unlike binary classification, in multiclass classification you have more than two classes in your dataset.

The technique of *one-vs-all* (or *one-vs-rest*) involves dividing your training set into multiple binary classification problems, rather than as a single multiclass classification problem.

For example, say you have three classes 1,2,3. Your first binary classifier will distinguish between class 1 and classes 2 and 3, your second binary classifier will distinguish between class 2 and classes 1 and 3, and your final binary classifier will distinguish between class 3 and classes 1 and 2.

Then to make the prediction, you pick the class i which maximizes $\max_i h_\theta^{(i)}(x)$.

9.17.2 Confusion Matrices

For classification, evaluation often comes in the form of a **confusion matrix**.

The core values are:

- **True positives (TP)**: samples classified as positive which were labeled positive
- **True negatives (TN)**: samples classified as negative which were labeled negative
- **False positives (FP)**: samples classified as positive which were labeled negative
- **False negatives (FN)**: samples classified as negative which were labeled positive

A few other metrics are computed from these values:

- **Accuracy:** How often is the classifier correct? ($\frac{TP+TN}{\text{total}}$)
- **Misclassification rate** (or “**error rate**”): How often is the classifier wrong? ($\frac{FP+FN}{\text{total}} = 1 - \text{accuracy}$)
- **Recall** (or “**sensitivity**” or “**true positive rate**”): How often are positive-labeled samples predicted as positive? ($\frac{TP}{\text{num positive-labeled examples}}$)
- **False positive rate:** How often are negative-labeled samples predicted as positive? ($\frac{FP}{\text{num negative-labeled examples}}$)
- **Specificity** (or “**true negative rate**”): How often are negative-labeled samples predicted as negative? ($\frac{TN}{\text{num negative-labeled examples}}$)
- **Precision:** How many of the predicted positive samples are correctly predicted? ($\frac{TP}{TP+FP}$)
- **Prevalence:** How many labeled-positive samples are there in the data? ($\frac{\text{num positive-labeled examples}}{\text{num examples}}$)

Some other values:

- **Positive predictive value (PPV):** precision but takes prevalence into account. With a perfectly balanced dataset (i.e. equal positive and negative examples, that is prevalence is 0.5), the PPV equals the precision.
- **Null error rate:** how often you would be wrong if you just predicted positive for every example. This is a good starting baseline metric to compare your classifier against.
- **F-score:** The weighted average of recall and precision
- **Cohen's Kappa:** a measure of how well the classifier performs compared against if it had just guessed randomly, that is a high Kappa score happens when there is a big difference between the accuracy and the null error rate.
- **ROC Curve:** (see the section on this)

References

- <http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/>

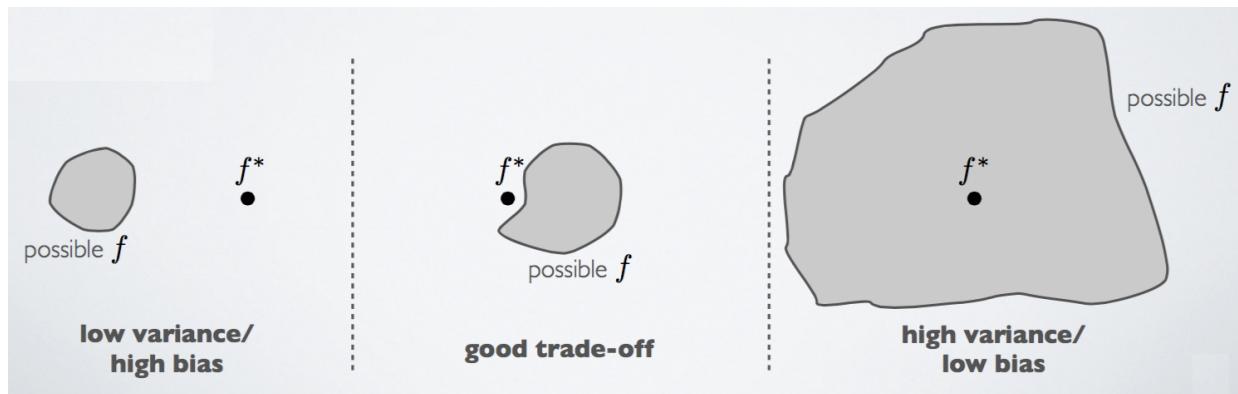
9.18 Overfitting

Overfitting is a problem where your hypothesis describes the training data *too well*, to the point where it cannot generalize to new examples. It is a *high variance* problem. In contrast, *underfitting* is a *high bias* problem.

To clarify, if your model has no bias, it means that it makes no errors on your training data (i.e. it does not underfit). If your model has no variance, it means your model generalizes well on your test data (i.e. it does not overfit). It is possible to have bias and variance problems simultaneously.

Another way to think of this is that:

- variance = how much does the model vary if the training data changes? I.e. what space of possible models does this cover?



Bias and Variance

- bias = is the average model close to the “true” solution/model?

There is a *bias-variance trade-off*, in which improvement of one is typically at the detriment of the other.

You can think of generalization error as the sum of bias and variance. You want to keep both low, if possible.

Overfitting can happen if your hypothesis is too complex, which can happen if you have too many features. So you will want to through a *feature selection* phase and pick out features which seem to provide the most value.

Alternatively, you can use the technique of *regularization*, in which you keep all your features, but reduce the magnitudes/values of parameters θ_j . This is a good option all of your features are informative and you don't want to get rid of any.

9.19 Regularization

The intuition behind regularization is that, if you have small values for your parameters $\theta_0, \theta_1, \dots, \theta_n$, then you have a “simpler” hypothesis which is less prone to overfitting.

In practice, there may be many combination of parameters which fit your data well. However, some may overfit/not generalize well. We want to introduce some means of valuing these simpler hypotheses over more complex ones (i.e. with larger parameters). We can do so with regularization.

So generally regularization is about shrinking your parameters to make them smaller. For linear regression, you accomplish this by modifying the cost function to include the term $\lambda \sum_{j=1}^n \theta_j^2$ at the end:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2$$

Note that we are not shrinking θ_0 . In practice, it does not make much of a difference if you include it or not; standard practice is to leave it out.

λ here is called the *regularization parameter*. It tunes the balance between fitting the data and keeping the parameters small (i.e. each half of the cost function). If you make λ *too* large for your problem, you may make your parameters too close to 0 for them to be meaningful.

The additional $\lambda \sum_{j=1}^n \theta_j^2$ term is called the *regularization loss*, and the rest of the loss function is called the *data loss*.

References

- Andrew Ng's Machine Learning Coursera course
- <https://cs231n.github.io/linear-classify/>

9.19.1 Regularized Linear Regression

We can update gradient descent to work with our regularization term:

$$\begin{aligned}\theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\ \theta_j &:= \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} + \frac{\lambda}{m} \theta_j \\ j &= (1, 2, 3, \dots, n)\end{aligned}$$

The θ_j part can be re-written as:

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

If we are using the normal equation, we can update it to a regularized form as well:

$$\theta = (X^T X + \lambda M)^{-1} X^T y$$

Where M is an $n+1 \times n+1$ matrix, where the diagonal is all ones, except for the element at $(0, 0)$ which is 0, and every other element is also 0.

9.19.2 Regularized Logistic Regression

We can also update the logistic regression cost function with the regularization term:

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

Then we can update gradient descent with the new derivative of this cost function for the parameters θ_j where $j \neq 0$

$$\begin{aligned}\theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\ \theta_j &:= \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} + \frac{\lambda}{m} \theta_j \\ j &= (1, 2, 3, \dots, n)\end{aligned}$$

It looks the same as the one for linear regression, but again, the actual hypothesis function h_θ is different.

9.20 Discriminative vs Generative learning algorithms

Discriminative learning algorithms include algorithms like logistic regression, decision trees, kNN, and SVM. Discriminative approaches try to find some way of separating data (*discriminating* them), such as in logistic regression which tries to find a dividing line and then sees where new data lies in relation to that line.

Say our input features are x and y is the class.

Discriminative learning algorithms learn $P(y|x)$ directly, that is it tries to learn the probability of y directly as a function of x . To put it another way, what is the probability this new data is of class y given the features x ?

Generative learning algorithms instead tries to develop a model for each class and sees which model new data conforms to.

Generative learning algorithms learn $P(x|y)$ and $P(y)$ instead (that is, it models the joint distribution $P(x, y)$). So instead they ask, if this were class y , what is the probability of seeing these new features x ? You're basically trying to figure out what class is most likely to have *generated* the given features x .

$P(y)$ is the class prior/the prior probability of seeing the class y , that is the probability of class y if you don't have any other information.

It is easier to estimate the conditional distribution $P(y|x)$ than it is the joint distribution $P(x, y)$, though generative models can be much stronger. With $P(x, y)$, it is easy to calculate the same conditional ($P(y|x) = \frac{P(x,y)}{P(x)}$).

For both discriminative and generative approaches, you will have parameters and latent variables θ which govern these distributions. We treat θ as a random variable.

9.21 Neural Networks

9.21.1 Neural networks and overfitting

You could fit a small neural network:

- fewer parameters, more prone to underfitting
- computationally cheaper

Or a large neural network:

- more parameters, more prone to overfitting
- computationally more expensive
- use regularization (λ) to address overfitting

Generally, large neural networks with regularization performs better than a smaller neural network.

9.21.2 Determining the number of hidden layers to use

When you can do is just split your data into training/test/cross validation and see how performance changes with the number of hidden layers. Pick one that looks promising when run on your cross validation data, then see how it generalizes to the test data.

9.22 Metrics

9.22.1 Area Under Curve (AUC)

AUC is a metric for binary classification and is especially useful when dealing with *high-bias* data, that is, where one class is much more common than the other. Using accuracy as a metric falls apart in high-bias datasets: for example, say you have 100 training examples, one of which is positive, the rest of which are negative. You could develop a model which just labels every thing negative, and it would have 99% accuracy. So accuracy doesn't really tell you enough here.

Many binary classifiers output some continuous value (0-1), rather than class labels; there is some threshold (usually 0.5) above which one label is assigned, and below which the other label is assigned. Some models may work best with a different threshold. Changing this threshold leads to a trade off between true positives and false positives - for example, decreasing the threshold will yield more true positives, but also more false positives.

AUC runs over all thresholds and plots the true vs false positive rates. This curve is called a *receiver operating characteristic* curve, or *ROC* curve. A random classifier would give you equal false and true positives, which leads to a AUC of 0.5; the curve in this case would be a straight line. The better the classifier is, the more area under the curve there is (so the AUC approaches 1).

9.23 Strategies for Applying Machine Learning

9.23.1 What if your algorithm doesn't perform well? What should you try next?

- Get more training examples (can help with high variance problems)
- Try smaller sets of features (can help with high variance problems)
- Try additional features (can help with high bias problems)
- Try adding polynomial features (x_1^2, x_2^2, x_1x_2 , etc) (can help with high bias problems)
- Try decreasing the regularization parameter λ (can help with high bias problems)
- Try increasing the regularization parameter λ (can help with high variance problems)

But how do you decide which thing to try?

9.23.2 Machine learning diagnostics

In machine learning, a diagnostic is:

A test that you can run to gain insight [about] what is/isn't working with a learning algorithm, and gain guidance as to how best to improve its performance.

They take time to implement but can save you a lot of time by preventing you from going down fruitless paths.

9.23.3 Evaluating a hypothesis

You can't evaluate a hypothesis with the cost function because minimizing the error can lead to overfitting.

A good approach is to take your data and split it randomly into a training set and a test set (e.g. a 70%/30% split). Then you train your model on the training set and see how it performs on the test set.

For linear regression, you might do things this way:

- Learn parameter θ from training data by minimizing training error $J(\theta)$.
- Compute test set error (using the squared error) (m_{test} is the test set size):

$$J_{\text{test}}(\theta) = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} (h_{\theta}(x_{\text{test}}^{(i)}) - y_{\text{test}}^{(i)})^2$$

For logistic regression, you might do things this way:

- Learn parameter θ from training data by minimizing training error $J(\theta)$.
- Compute test set error (m_{test} is the test set size):

$$J_{\text{test}}(\theta) = -\frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} y_{\text{test}}^{(i)} \log h_{\theta}(x_{\text{test}}^{(i)}) + (1 - y_{\text{test}}^{(i)}) \log h_{\theta}(x_{\text{test}}^{(i)})$$

- Alternatively, you can use the misclassification error (“0/1 misclassification error”, read “zero-one”), which is just the fraction of examples that your hypothesis has mislabeled:

$$\begin{aligned} \text{err}(h_{\theta}(x), y) &= \begin{cases} 1, & \text{if } h_{\theta}(x) \geq 0.5, y = 0 \text{ or if } h_{\theta}(x) < 0.5, y = 1 \\ 0, & \text{otherwise} \end{cases} \\ \text{test error} &= \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \text{err}(h_{\theta}(x_{\text{test}}^{(i)}), y_{\text{test}}^{(i)}) \end{aligned}$$

A better way of splitting the data is to not split it only into training and testing sets, but to also include a *cross validation* set. A typical ratio is 60% training, 20% cross validation, 20% testing.

So instead of just measuring the test error, you would also measure the cross validation error.

This is useful if you are trying to determine what features to include for your model. You can try different hypotheses and then see how they perform on your cross validation set. If they seem promising, you can then test your model’s generalizability on your test set. In this way, you can better avoid overfitting.

You can use these errors to identify what kind of problem you have if your model isn’t performing well:

- If your training error is large and your cross validation/test set error is large, then you have a high bias (underfitting) problem.
- If your training error is small and your cross validation/test set error is large, then you have a high variance (overfitting) problem.

9.23.4 Choosing a good λ for regularization

We can choose some range of values for λ and evaluate the performance for each on the cross validation set. Pick the best one and then see how it generalizes on the test set.

Just remember that large values of lambda can lead to underfitting problems (since the parameters get close to 0), and small values of lambda can lead to overfitting.

9.23.5 Learning curves

To generate a learning curve, you deliberately shrink the size of your training set and see how the training and cross validation errors change as you increase the training set size. This way you can see how your model improves (or doesn't, if something unexpected is happening) with more training data.

With smaller training sets, we expect the training error will be low because it will be easier to fit to less data. So as training set size grows, the average training set error is expected to grow. Conversely, we expect the average cross validation error to decrease as the training set size increases.

If it seems like the training and cross validation error curves are flattening out at a high error as training set size increases, then you have a high bias problem. The curves flattening out indicates that getting more training data will not (by itself) help much.

On the other hand, high variance problems are indicated by a large gap between the training and cross validation error curves as training set size increases. You would also see a low training error. In this case, the curves are converging and more training data would help.

9.24 Machine Learning System Design

Before you start building your machine learning system, you should:

- Be explicit about the problem.
- Brainstorm some possible strategies.
 - What features might be useful?
 - Do you need to collect more data?

Then to start:

- Start with a simple algorithm which can be implemented quickly.
- Test the simple algorithm on your cross-validation data.
- Plot learning curves to decide where things need work:
 - Do you need more data?
 - Do you need more features?
 - And so on.
- Error analysis: manually examine the examples in the cross validation set that your algorithm made errors on. Try to identify patterns in these errors. Are there categories of examples that the model is failing on in particular? Are there any other features that might help?

If you have an idea for a feature which may help, it's best to just test it out. This process is much easier if you have a single metric for your model's performance. You can use cross-validation error or others, mentioned below.

When it comes to *skewed classes* (or *high bias data*), metric selection is more nuanced.

For instance, say you have a dataset where only 0.5% of the data is in category 1 and the rest is in category 0. You run your model and find that it categorized 99.5% of the data correctly! But because of the skew in that data, your model could just be: classify each example in category 0, and it would achieve that accuracy.

Note that the convention is to set the rare class to 1 and the other class to 0. That is, we try to predict the rare class.

Instead, you may want to use *precision/recall* as your evaluation metric.

| | 1T | 0T |
|----|----------------|----------------|
| 1P | True positive | False positive |
| 0P | False negative | True negative |

Where 1T/0T indicates the actual class and 1P/0P indicates the predicted class.

Precision is the number of true positives over the total number predicted as positive. That is, what fraction of the examples labeled as positive actually are positive?

$$\frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

Recall is the number of true positives over the number of actual positives. That is, what fraction of the positive examples in the data were identified?

$$\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

So in the previous example, our simple classifier would have a recall of 0.

There is a trade-off between precision and recall.

Say you are using a logistic regression model for this classification task. Normally, the category threshold in logistic regression is 0.5, that is, predict class 1 if $h_\theta(x) \geq 0.5$ and predict class 0 if $h_\theta(x) < 0.5$.

But you may want to only classify an example as 1 if you're very confident. So you may change the threshold to 0.9 to be stricter about your classifications. In this case, you would increase precision, but lower recall since the model may not be confident enough about some of the more ambiguous positive examples.

Conversely, you may want to lower the threshold to avoid false negatives, in which case recall increases, but precision decreases.

So how do you compare precision/recall values across algorithms to determine which is best? You can condense precision and recall into a single metric: the F_1 score (also just called the *F score*, which is the harmonic mean of the precision and recall):

$$F_1\text{score} = 2 \frac{PR}{P + R}$$

Although more data doesn't always help, it very often does. Many algorithms perform significantly better as they get more and more data. Even relatively simple algorithms can outperform more sophisticated ones, solely on the basis of having more training data.

9.25 Unsupervised Learning

In unsupervised learning, our data does not have any labels. Unsupervised learning algorithms try to find some structure in the data.

9.25.1 K-Means Clustering Algorithm

First, randomly initialize K points, called the *cluster centroids*.

Then iterate:

- Cluster assignment step: go through each data point and assign it to the closest of the K centroids.
- Move centroid step: move the centroids to the average of their points.

Closeness is computed by some distance metric, e.g. euclidean.

More formally, there are two inputs:

- K - the number of clusters
- The training set
 $x^{(1)}, x^{(2)}, \dots, x^{(m)}$

Where $x^{(i)} \in \mathbb{R}^n$ (we drop the $x_0 = 1$ convention).

Randomly initialize K cluster centroids $\mu_1, \mu_2, \dots, \mu_K \in \mathbb{R}^n$.

Repeat:

- For $i = 1$ to m
 - $c^{(i)} := \text{index (from 1 to } K \text{) of cluster centroid closest to } x^{(i)}$. That is, $c^{(i)} := \min_k \|x^{(i)} - \mu_k\|$.
- For $k = 1$ to K
 - $\mu_k := \text{average (mean) of points assigned to cluster } k$

If you have an empty cluster, it is common to just eliminate it entirely.

We can denote the cluster centroid of the cluster to which example $x^{(i)}$ has been assigned as $\mu_{c^{(i)}}$.

In K-means, the optimization objective is:

$$J(c^{(i)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^m \|x^{(i)} - \mu_{c^{(i)}}\|^2$$

$$\min_{c^{(i)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K} J(c^{(i)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$$

This cost function is sometimes called the *distortion* cost function or the distortion of the K-means algorithm.

The algorithm outlined above is minimizing the cost: the first step tries to minimize $c^{(i)}, \dots, c^{(m)}$ and the second step tries to minimize μ_1, \dots, μ_K .

One question is - what's the best way to initialize the initial centroids to avoid local minima of the cost function?

First of all, you should halve $K < m$ (i.e. less than your training examples.)

Then randomly pick K training examples. Use these as your initialization points (i.e. set μ_1, \dots, μ_K to these K examples).

Then, to better avoid local optima, just rerun K-means several times (e.g. 50-1000 times) with new initializations of points. Keep track of the resulting cost function and then pick the clustering that gave the lowest cost.

So, how do you choose a good value for K ?

Unfortunately, there is no good way of doing this automatically. The most common way is to just choose it manually by looking at the output. If you plot out the data and look at it - even among people it is difficult to come to a consensus on how many clusters there are.

One method that some use is the *Elbow method*. In this approach, you vary K , run K-means, and compute the cost function for each value. If you plot out K vs the cost functions, there may be a clear "elbow" in the graph and you pick the K at the elbow. However, most of the time there isn't a clear elbow.

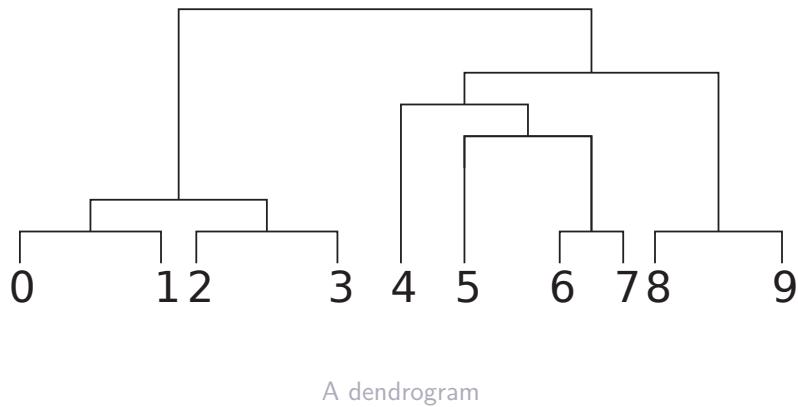
9.25.2 Hierarchical Agglomerative Clustering

Hierarchical agglomerative clustering (HAC) is a bottom-up clustering process which is fairly simple:

1. Find two closest data points or clusters, merge into a cluster (and remove the original points or clusters which formed the new cluster)
2. Repeat

This results in a hierarchy (e.g. a tree structure) describing how the data can be grouped into clusters and clusters of clusters. This structure can be visualized as a dendrogram:

Two things which must be specified for HAC are:



- the distance metric: euclidean, cosine, etc
- the merging approach - that is, how is the distance between two clusters measured?
 - *complete linkage* - use the distance between the two further points
 - *average linkage* - take the average distances of all pairs between the clusters
 - *single linkage* - take the distance between the two nearest points
 - (there are others as well)

Unlike K-means, HAC is deterministic (since there are no randomly-initialized centroids) but it can be unstable: changing a few points or the presence of some outliers can vastly change the result. Scaling of variables/features can also affect clustering.

9.26 Support Vector Machines

SVMs can be powerful for learning non-linear functions and are widely-used.

With SVMs, the optimization objective is:

$$\min_{\theta} \sum_{i=1}^m [y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)})] + \frac{\lambda}{2} \sum_{j=1}^n \theta_j^2$$

Where the term at the end is the regularization term. Note that this is quite similar to the objective function for logistic regression; we have just removed the $\frac{1}{m}$ term (removing it does not make a difference to our result because it is a constant) and substituted the log hypothesis terms for two new cost functions.

If we break up the logistic regression objective function into terms (that is, the first sum and the regularization term), we might write it as $A + \lambda B$.

The SVM objective is often instead notated by convention as $CA + B$. You can think of C as $\frac{1}{\lambda}$. That is, where increasing λ brings your parameters closer to zero, the regularization parameter C has the opposite effect - as it grows, so do your parameters, and vice versa.

With that representation in mind, we can rewrite the objective by replacing the λ with C on the first term:

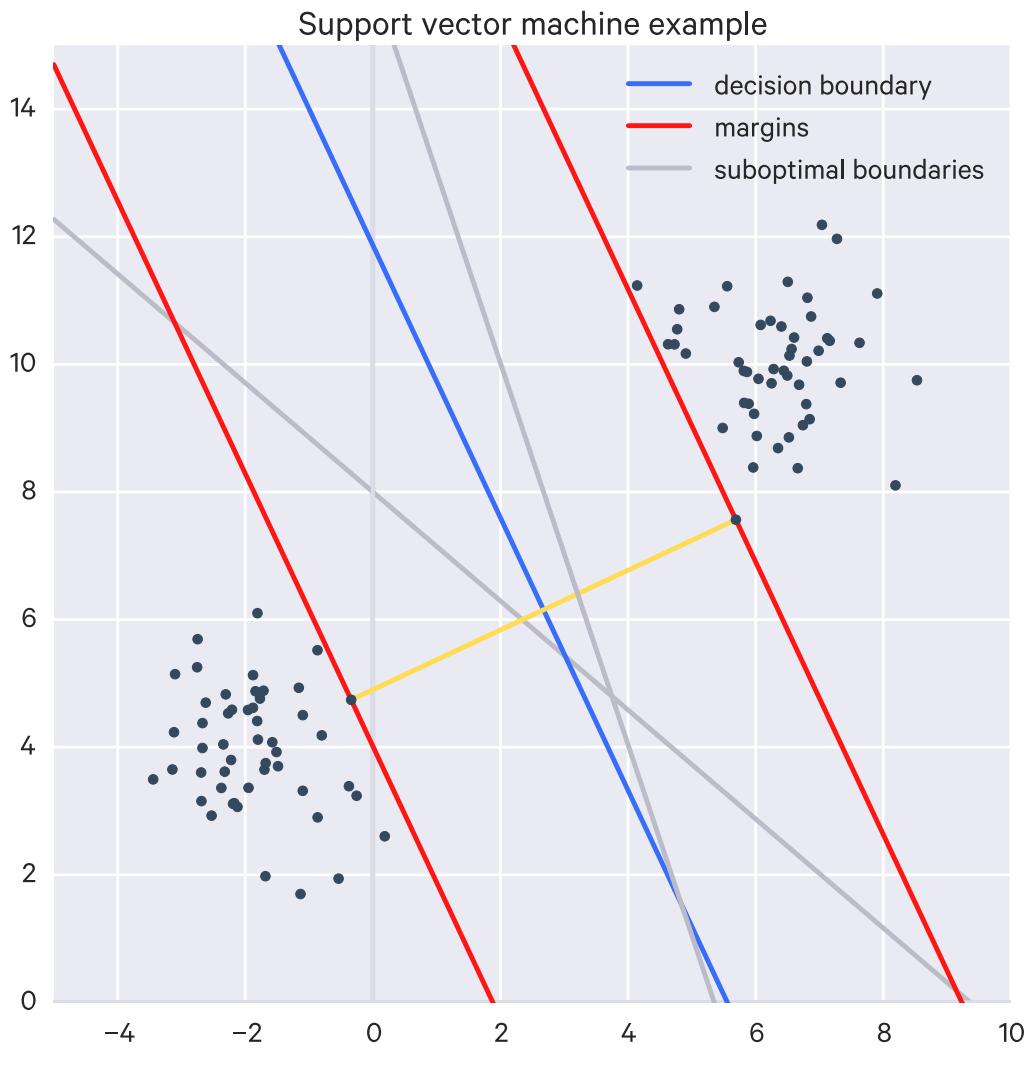
$$\min_{\theta} C \sum_{i=1}^m [y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)})] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

The SVM hypothesis is:

$$h_{\theta}(x) = \begin{cases} 1 & \text{if } \theta^T x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

SVMs are sometimes called *large margin* classifiers.

Take the following data:



SVM and margins

On the left, a few different lines separating the data are drawn. The optimal one found by SVM is the one in orange. It is the optimal one because it has the largest margins, illustrated by the red lines

on the right (technically, the margin is orthogonal from the decision boundary *to* those red lines). When C is very large, SVM tries to maximize these margins.

However, outliers can throw SVM off if your regularization parameter C is too large, so in those cases, you may want to try a smaller value for C .

9.26.1 Kernels

Kernels are the main technique for adapting SVMs to do complex non-linear classification.

A note on notation. Say your hypothesis looks something like:

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \dots$$

We can instead notate each non-parameter term as a feature f , like so:

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + \theta_4 f_4 + \dots$$

For SVMs, how do we choose these features?

What we can do is compute features based on x 's proximity to landmarks $l^{(1)}, l^{(2)}, l^{(3)}, \dots$. For each landmark, we get a feature:

$$f_i = \text{similarity}(x, l^{(i)}) = \exp\left(-\frac{\|x - l^{(i)}\|^2}{2\sigma^2}\right)$$

Here, the $\text{similarity}(x, l^{(i)})$ function is the *kernel*, sometimes just notated $k(x, l^{(i)})$.

We have a choice in what kernel function we use; here we are using *Gaussian kernels*. In the Gaussian kernel we have a parameter σ .

If x is close to $l^{(i)}$, then we expect $f_i \approx 1$. Conversely, if x is far from $l^{(i)}$, then we expect $f_i \approx 0$.

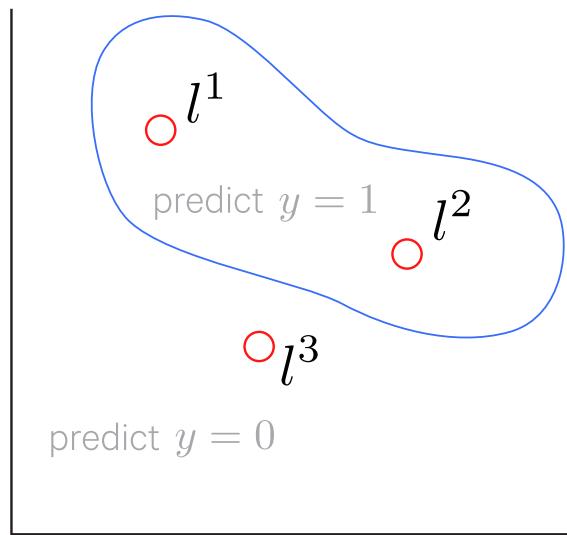
With this approach, classification becomes based on distances to the landmarks - points that are far away from certain landmarks will be classified 0, points that are close to certain landmarks will be classified 1. And thus we can get some complex decision boundaries like so:

So how do you choose the landmarks?

You can take each training example and place a landmark there. So if you have m training examples, you will have m landmarks.

So given $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$, choose $l^{(1)} = x^{(1)}, l^{(2)} = x^{(2)}, \dots, l^{(m)} = x^{(m)}$.

Then given a training example $(x^{(i)}, y^{(i)})$, we can compute a feature vector f , where $f_0 = 1$, like so:



An example SVM decision boundary

$$f = \begin{bmatrix} f_0 = 1 \\ f_1^{(i)} = \text{sim}(x^{(i)}, l^{(1)}) \\ f_2^{(i)} = \text{sim}(x^{(i)}, l^{(2)}) \\ \vdots \\ f_i^{(i)} = \text{sim}(x^{(i)}, l^{(i)}) \\ \vdots \\ f_m^{(i)} = \text{sim}(x^{(i)}, l^{(m)}) \end{bmatrix}$$

Then instead of x we use our feature vector f . So our objective function becomes:

$$\min_{\theta} C \sum_{i=1}^m [y^{(i)} \text{cost}_1(\theta^T f^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T f^{(i)})] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

Note that here $n = m$ because we have a feature for each of our m training examples.

Of course, using a landmark for each of your training examples makes SVM difficult on large datasets. There are some implementation tricks to make it more efficient, though.

When choosing the regularization parameter C , note that:

- A large C means lower bias, high variance
- A small C means higher bias, low variance

For the Gaussian kernel, we also have to choose the parameter σ^2 .

- A large σ^2 means that features f_i vary more smoothly. Higher bias, lower variance.
- A small σ^2 means that features f_i vary less smoothly. Lower bias, higher variance.

When using SVM, you also need to choose a kernel, which could be the Gaussian kernel, or it could be no kernel (i.e. a linear kernel), or it could be one of many others. The Gaussian and linear kernels are by far the most commonly used.

You may want to use a linear kernel if n is very large, but you don't have many training examples (m is small). Something more complicated may overfit if you only have a little data.

The Gaussian kernel is appropriate if n is small and/or m is large. Note that you should perform feature scaling before using the Gaussian kernel.

Not all similarity functions make valid kernels - they must satisfy a condition called Mercer's Theorem which allows the optimizations that most SVM implementations provide and also so they don't diverge.

Other off-the-shelf kernels include:

- Polynomial kernel: $k(x, l) = (x^T l)^2$, or $k(x, l) = (X^T l)^3$, or $k(x, l + 1)^3$, etc (there are many variations), the general form is $(x^T l + \text{constant})^{\text{degree}}$. It usually performs worse than the Gaussian kernel.
- More esoteric ones: String kernel, chi-square kernel, histogram intersection kernel, ...

But these are seldom, if ever, used.

Some SVM packages have a built-in multi-class classification functionality. Otherwise, you can use the one-vs-all method. That is, train K SVMs, one to distinguish $y = i$ from the rest, for $i = 1, 2, \dots, K$, then get $\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(K)}$, and pick class i with the largest $(\theta^{(i)})^T x$.

If n is large relative to m , e.g. $n = 10000, m \in [10, 1000]$, then it may be better to use logistic regression, or SVM without a kernel (linear kernel).

If n is small (1-1000) and m is intermediate (10-50000), then you can try SVM with the Gaussian kernel.

If n is small (1-1000) but m is large (50000+), then you can create more features and then use logistic regression or SVM without a kernel, since otherwise SVMs struggle at large training sizes.

SVM without a kernel works out to be similar to logistic regression for the most part.

Neural networks are likely to work well for most of these situations, but may be slower to train.

The SVM's optimization problem turns out to be convex, so good SVM packages will find global minimum or something close to it (so no need to worry about local optima).

Other rules of thumb:

- Use linear kernel when number of features is larger than number of observations.
- Use gaussian kernel when number of observations is larger than number of features.
- If number of observations is larger than 50,000 speed could be an issue when using gaussian kernel; hence, one might want to use linear kernel. [Source](#)

Also:

Usually, the decision is whether to use linear or an RBF (aka Gaussian) kernel. There are two main factors to consider:

Solving the optimisation problem for a linear kernel is much faster, see e.g. LI-BILINEAR. Typically, the best possible predictive performance is better for a nonlinear kernel (or at least as good as the linear one).

It's been shown that the linear kernel is a degenerate version of RBF, hence the linear kernel is never more accurate than a properly tuned RBF kernel. Quoting the abstract from the paper I linked:

The analysis also indicates that if complete model selection using the Gaussian kernel has been conducted, there is no need to consider linear SVM.

A basic rule of thumb is briefly covered in NTU's practical guide to support vector classification (Appendix C).

If the number of features is large, one may not need to map data to a higher dimensional space. That is, the nonlinear mapping does not improve the performance. Using the linear kernel is good enough, and one only searches for the parameter C.

Your conclusion is more or less right but you have the argument backwards. In practice, the linear kernel tends to perform very well when the number of features is large (e.g. there is no need to map to an even higher dimensional feature space). A typical example of this is document classification, with thousands of dimensions in input space.

In those cases, nonlinear kernels are not necessarily significantly more accurate than the linear one. This basically means nonlinear kernels lose their appeal: they require way more resources to train with little to no gain in predictive performance, so why bother.

TL;DR

Always try linear first since it is way faster to train (AND test). If the accuracy suffices, pat yourself on the back for a job well done and move on to the next problem. If not, try a nonlinear kernel. [Source](#)

more on support vector machines

Support vector machines is another way of coming up with decision boundaries to divide a space.

Here the decision boundary is positioned so that its *margins* are as wide as possible.

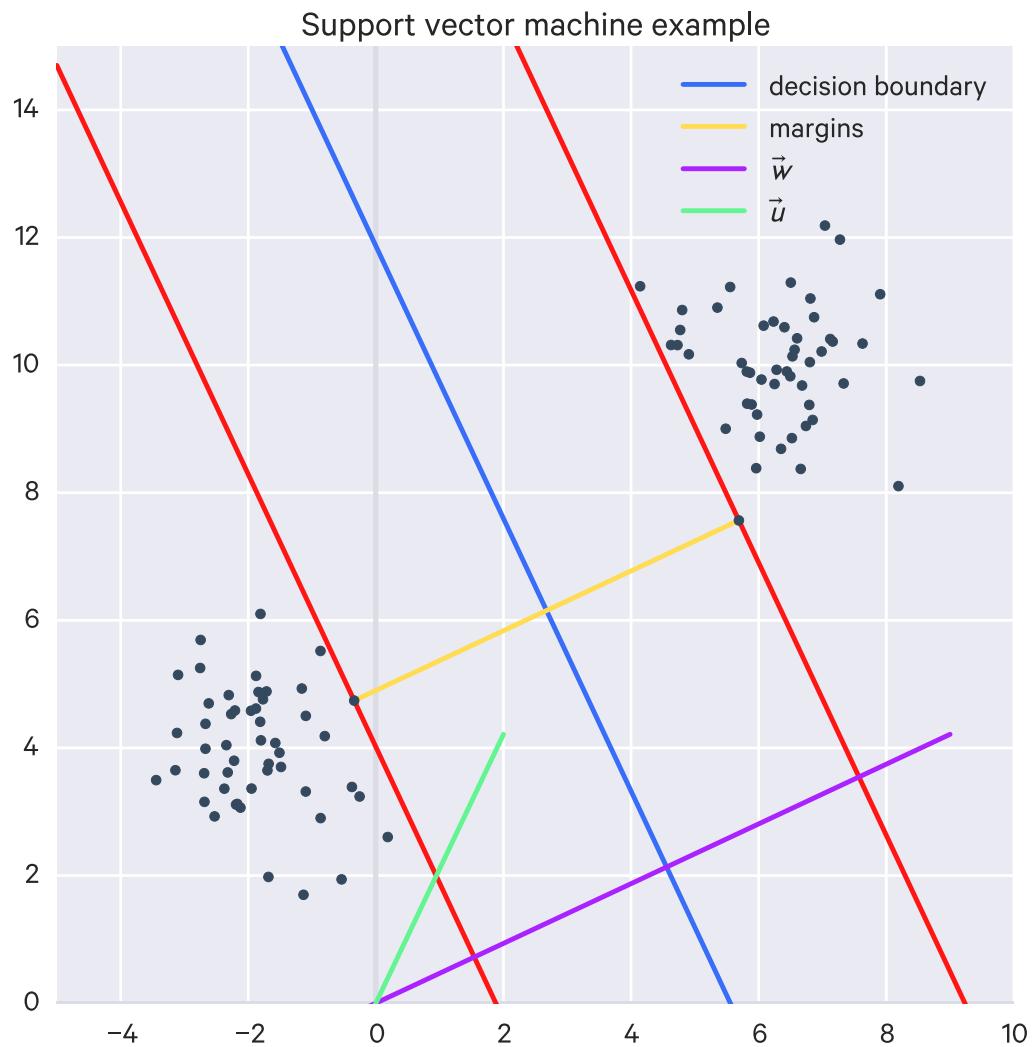
We can consider some vector \vec{w} which is perpendicular to the decision boundary and has an unknown length.

Then we can consider an unknown vector \vec{u} that we want to classify.

We can compute their dot product, $\vec{w} \cdot \vec{u}$, and see if it is greater than or equal to some constant c .

To make things easier to work with mathematically, we set $b = -c$ and rewrite this as:

$$\vec{w} \cdot \vec{u} + b \geq 0$$



Support Vector Machines

This is our decision rule: if this inequality is true, we have a positive example.

Now we will define a few things about this system:

$$\vec{w} \cdot \vec{x}_+ + b \geq 1$$

$$\vec{w} \cdot \vec{x}_- + b \leq -1$$

Where \vec{x}_+ is a positive training example and \vec{x}_- is a negative training example. So we will insist that these inequalities hold.

For mathematical convenience, we will define another variable y_i like so:

$$y_i = \begin{cases} y_i = +1 & \text{if positive example} \\ y_i = -1 & \text{if negative example} \end{cases}$$

So we can rewrite our constraints as:

$$\begin{aligned} y_i(\vec{w} \cdot \vec{x}_+ + b) &\geq 1 \\ y_i(\vec{w} \cdot \vec{x}_- + b) &\geq 1 \end{aligned}$$

Which ends up just collapsing into:

$$y_i(\vec{w} \cdot \vec{x} + b) \geq 1$$

Or:

$$y_i(\vec{w} \cdot \vec{x} + b) - 1 \geq 0$$

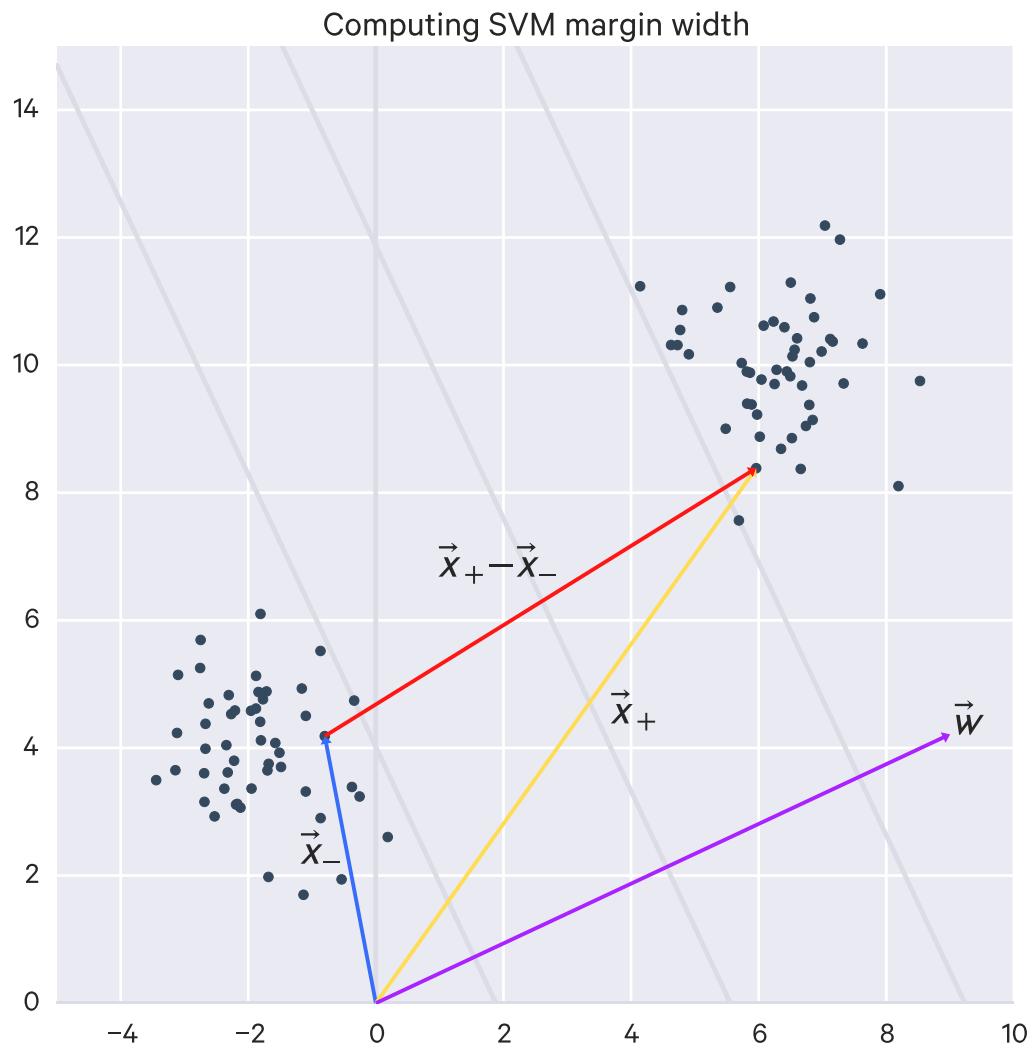
We then add an additional constraint for an x_i in the gutter (that is, within the margin of the decision boundary):

$$y_i(\vec{w} \cdot \vec{x} + b) - 1 = 0$$

So how do you compute the total width of the margins?

You can take a negative example \vec{x}_- and a positive example \vec{x}_+ and compute their difference $\vec{x}_+ - \vec{x}_-$. This resulting vector is not orthogonal to the decision boundary, so we can project it onto the unit vector \hat{w} (the unit vector of the \vec{w} , which is orthogonal to the decision boundary):

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



Support Vector Machines

Using our previous constraints we get $\vec{x}_+ = 1 - b$ and $-\vec{x}_- = 1 + b$, so the end result is:

$$\text{width} = \frac{2}{\|\vec{w}\|}$$

We want to maximize the margins, that is, we want to maximize the width, and we can divide by $\frac{1}{2}$ because we still have a meaningful maximum, and that in turn can be interpreted as the minimum of the length of \vec{w} , which we can rewrite in a more mathematically convenient form (and still have the same meaningful minimum):

$$\max\left(\frac{2}{\|\vec{w}\|}\right) \rightarrow \max\left(\frac{1}{\|\vec{w}\|}\right) \rightarrow \min(\|\vec{w}\|) \rightarrow \min\left(\frac{1}{2}\|\vec{w}\|^2\right)$$

Let's turn this into something we can maximize, incorporating our constraints. We have to use Lagrange multipliers which provide us with this new function we can maximize without needing to think about our constraints anymore:

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

(Note that the Lagrangian is an objective function which includes equality constraints).

Where L is the function we want to maximize, and the sum is the sum of the constraints, each with a multiplier α_i .

So then to get the maximum, we just compute the partial derivatives and look for zeros:

$$\begin{aligned} \frac{\partial L}{\partial \vec{w}} &= \vec{w} - \sum_i \alpha_i y_i \vec{x}_i = 0 \rightarrow \vec{w} = \sum_i \alpha_i y_i \vec{x}_i \\ \frac{\partial L}{\partial b} &= -\sum_i \alpha_i y_i = 0 \rightarrow \sum_i \alpha_i y_i = 0 \end{aligned}$$

Let's take these partial derivatives and re-use them in the original Lagrangian:

$$L = \frac{1}{2} \left(\sum_i \alpha_i y_i \vec{x}_i \right) \cdot \left(\sum_j \alpha_j y_j \vec{x}_j \right) - \sum_i \alpha_i y_i \vec{x}_i \cdot \left(\sum_j \alpha_j y_j \vec{x}_j \right) - \sum_i \alpha_i y_i b + \sum_i \alpha_i$$

Which simplifies to:

$$L = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \vec{x}_i \cdot \vec{x}_j$$

We see that this depends on $\vec{x}_i \cdot \vec{x}_j$.

Similarly, we can rewrite our decision rule, substituting for \vec{w} .

$$\vec{w} = \sum_i \alpha_i y_i \vec{x}_i$$

$$\vec{w} \cdot \vec{u} + b \geq 0$$

$$\sum_i \alpha_i y_i \vec{x}_i \cdot \vec{u} + b \geq 0$$

And similarly we see that this depends on $\vec{x}_i \cdot \vec{u}$.

The nice thing here is that this works in a convex space (proof not shown) which means that it cannot get stuck on a local maximum.

Sometimes you may have some training data \vec{X} which is not linearly separable. What you need is a transformation, $\phi(\vec{X})$ to take the data from its current space to a space where it *is* linearly separable.

Since the maximization and the decision rule depend only on the dot products of vectors, we can just substitute the transformation, so that:

- we want to maximize $\phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$
- for the decision rule, we have $\phi(\vec{x}_i) \cdot \phi(\vec{u})$

Since these are just dot products between the transformed vectors, we really only need a function which gives us that dot product:

$$K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$$

This function K is called the *kernel* function.

So if you have the kernel function, you don't even need to know the specific transformation - you just need the kernel function.

Some popular kernels:

- linear kernel: $K(\vec{u}, \vec{v}) = (\vec{u} \cdot \vec{v} + 1)^n$
- radial basis kernel: $e^{-\frac{\|\vec{x}_i - \vec{x}_j\|}{\sigma}}$

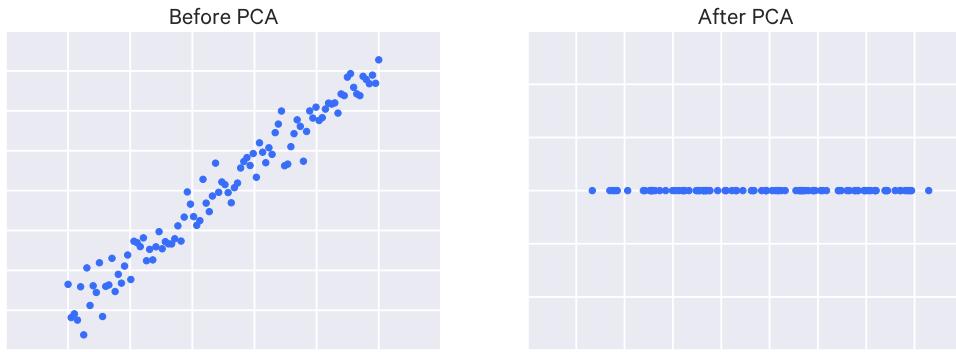
9.27 Dimensionality Reduction

9.27.1 Motivation: Data Compression

Sometimes some of your features may be redundant. You can combine these features in such a way that you project your higher dimension representation into a lower dimension representation while minimizing information loss. With the reduction in dimensionality, your algorithms will run faster.

9.27.2 Principal Component Analysis (PCA)

Say you have some data. This data has two dimensions, but you could more or less capture it in one dimension:

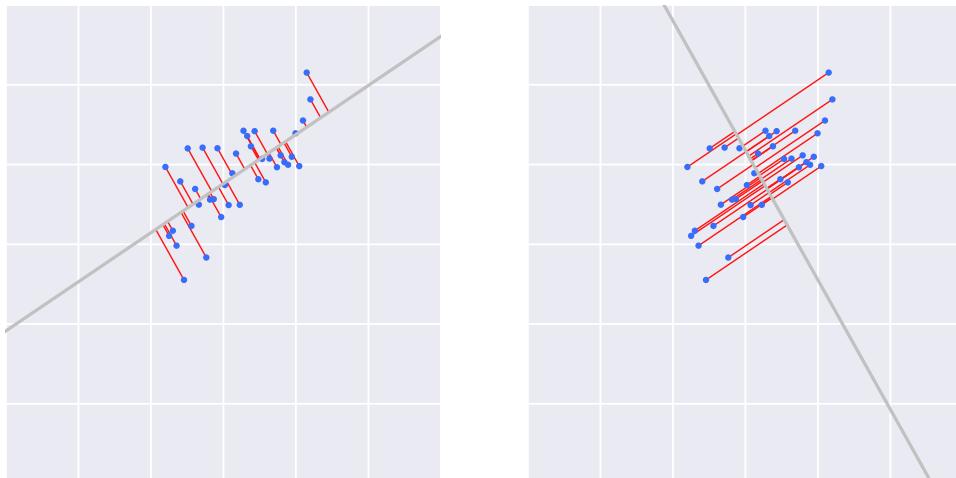


Reducing data dimensionality with PCA

Most of the variability of the data happens along that axis.

This is basically what PCA does.

PCA is the most commonly used algorithm for dimensionality reduction. PCA tries to identify a lower-dimensional surface to project the data onto such that the square *projection error* is minimized.



PCA example

PCA might project the data points onto the green line on the left. The projection error are the blue lines. Compare to the line on the right - PCA would not project the data onto that line since the projection error is much larger for that line.

This example is going from 2D to 1D, but you can use PCA to project from any n -dimension to a lower k -dimension. Using PCA, we find some k vectors and project our data onto the linear subspace spanned by this set of k vectors.

Note that this is different than linear regression, though the example might look otherwise. In PCA, the projection error is orthogonal to the line in question. In linear regression, it is vertical to the line. Linear regression also favors the target variable y whereas PCA makes no such distinction.

Prior to PCA you should perform mean normalization (i.e. ensure every feature has zero mean) on your features and scale them.

First you compute the *covariance matrix*, which is denoted Σ (same as summation, unfortunately):

$$\Sigma = \frac{1}{m} \sum_{i=1}^n (x^{(i)})(x^{(i)})^T$$

Then, you compute the *eigenvectors* of the matrix Σ using *singular value decomposition*:

$$[U, S, V] = \text{svd}(\Sigma)$$

The resulting U matrix will be an $n \times n$ orthogonal matrix which provides the projected vectors you're looking for, so take the first k column vectors of U . This $n \times k$ matrix can be called U_{reduce} , which you then transpose to get these vectors as rows, resulting in a $k \times n$ matrix which you then multiply by your feature matrix.

So how do you choose k , the number of principal components?

One way to choose k is so that most of the variance is retained.

If the average squared projection error (which is what PCA tries to minimize) is:

$$\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{\text{approx}}^{(i)}\|^2$$

And the total variation in the data is given by:

$$\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2$$

Then you would choose the smallest value of k such that:

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{\text{approx}}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01$$

That is, so that 99% of variance is retained.

This procedure for selecting k is made much simpler if you use the S matrix from the $\text{svd}(\Sigma)$ function.

The S matrix's only non-zero values are along its diagonal, $S_{11}, S_{22}, \dots, S_{nn}$. Using this you can instead just calculate:

$$1 - \frac{\sum_{i=1}^k S_{ii}}{\sum_{i=1}^n S_{ii}} \leq 0.01$$

Or, to put it another way:

$$\frac{\sum_{i=1}^k S_{ii}}{\sum_{i=1}^n S_{ii}} \geq 0.99$$

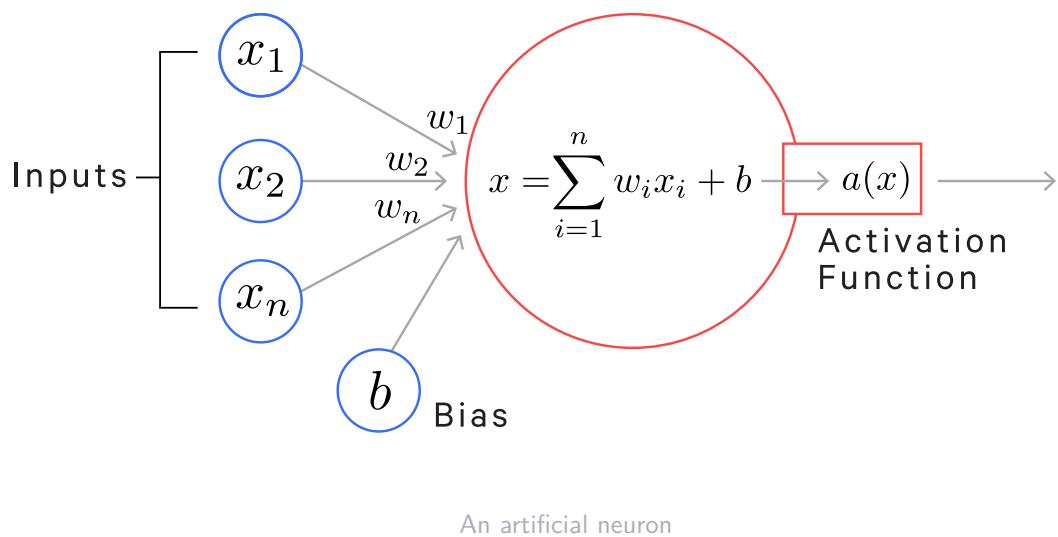
In practice, you can reduce the dimensionality quite drastically, such as by 5 or 10 times, such as from 10,000 features to 1,000, and retain variance.

But you should not use PCA prematurely - first try an approach without it, then later you can see if it helps.

9.28 Neural Networks

Neural networks are good at learning complex non-linear hypotheses in very large feature spaces.

In a neural network, a neuron is modeled as a *logistic unit*.



Our hypothesis function is $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$. This is the neuron's *sigmoid (logistic) activation function*.

Sometimes an additional input unit x_0 is included, which is called the *bias unit*, but this is always equal to 1 and thus sometimes omitted in diagrams.

Many neurons are arranged into layers, one after the other. The first layer is called the *input layer*, the last layer is called the *output layer*, and the layers in between are the *hidden layers*. The way these layers are arranged and structured is called the *architecture* of the neural network. This layer structure allows simpler networks to be aggregated to perform much more complex operations.

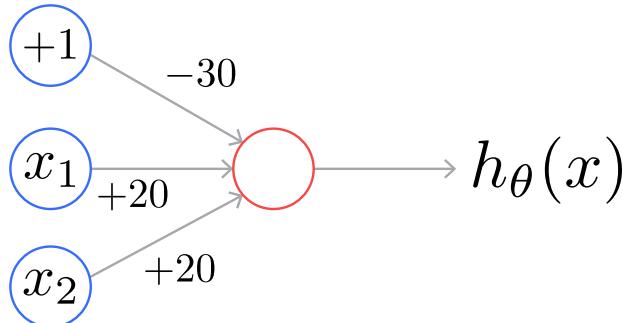
We denote the activation of neuron (unit) i in layer j by $a_i^{(j)}$. We also denote $\Theta^{(j)}$ as the weight matrix (parameters) controlling the function mapping from layer j to layer $j + 1$.

If a network has s_j units in layer j , s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimensions $s_{j+1} \times s_j + 1$.

With *forward propagation*, you compute the activations of the input units then forward propagate that to the activations of the hidden layers and so on until you reach the output layer.

Neural networks function kind of like logistic regression except they learn their own features based on the input features via the hidden layers.

As a simple example, let's say we want a neural network which can compute AND, and it learns the weights in the figure (remember, the $+1$ input is the bias unit).



A simple AND neural network

Here our hypothesis function would be:

$$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$$

Where g is the sigmoid function.

This yields the following table:

| x_1 | x_2 | $h_\Theta(x)$ |
|-------|-------|--------------------|
| 0 | 0 | $g(-30) \approx 0$ |
| 0 | 1 | $g(-10) \approx 0$ |
| 1 | 0 | $g(-10) \approx 0$ |
| 1 | 1 | $g(10) \approx 1$ |

Which is what we would want out of AND.

You can accomplish multiclass classification with neural networks by using a one-vs-all approach. Your output layer would have multiple units, one for each class, and the classification is the unit with the strongest response.

We will use the following notation:

- L = the total number of layers in the network
- s_l = the total number of units (excluding the bias unit) in layer l

- $K =$ the number of units in the output layer

The cost function for the neural network is a generalization of the one used for logistic regression. But instead of one logistic regression output unit, we instead have K of them.

$$(h_{\Theta}(x))_i = i^{\text{th}} \text{output}$$

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_{\Theta}(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

So basically we are summing logistic regression cost functions over our k output units with a regularization term.

So we want to find Θ to minimize this function. That is, we want $\min_{\Theta} J(\Theta)$. We can use gradient descent (or one of the other optimization algorithms, but here we are using gradient descent), so we need to compute the partial derivative terms of $J(\Theta)$, that is:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$$

We accomplish this optimization with the *backpropagation algorithm*.

The intuition in backpropagation is that $\delta_j^{(l)}$ is the “error” of node j in layer l . After you have completed forward propagation, you have these error terms for your output layer. Then you calculate the error terms for the earlier layers in the network. Proof not shown, but this comes down to:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$

In this case we are assuming $\lambda = 0$ but we will come back to that.

The formalization of the backpropagation algorithm:

For a training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, set $\Delta_{ij}^{(l)} = 0$ for all l, i, j .

Then, for $i = 1$ to m (that is, for each training example), do:

- Set $a^{(1)} = x^{(i)}$
- Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$
- Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$
- Then:

$$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

After completing those iterations, you then compute:

$$\begin{aligned} D_i^{(l)} j &:= \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0 \\ D_i^{(l)} j &:= \frac{1}{m} \Delta_{ij}^{(l)} \text{ if } j = 0 \end{aligned}$$

With gradient descent or other optimization algorithms you need to pick some initial values for the parameters Θ . So what do you initialize it to? For logistic regression, initializing the parameters to 0 works, but this is not the case with neural networks; if you initialize to 0, your weights will not change and just end up the same, which isn't what you want.

So you can initialize each of parameters to some random value in $[-\epsilon, \epsilon]$, where ϵ is a very small value.

So how do you decide on your neural network architecture? The number of your input units is just the dimensions of your features $x^{(i)}$. The number of your output units is Just the number of classes. For the number of hidden layers, a reasonable default is to just use one hidden layer. If you end up using more than one hidden layer, you can have the same number of hidden units in every layer. Usually the more units in each layer, the better.

So overall, the steps to training a neural network are:

1. Randomly initialize weights to small values near zero
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute the cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
 - Perform forward propagation and backpropagation using each training example and compute your Δ terms
 - Then compute $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
5. Use gradient descent or some other optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

9.29 Large Scale Machine Learning

9.29.1 Map Reduce

You can distribute the workload across computers to reduce training time.

For example, say you're running batch gradient descent with $b = 400$.

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^4 00_i (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

You can divide up (map) your batch so that different machines calculate the error of a subset (e.g. with 4 machines, each machine takes 100 examples) and then those results are combined (reduced/summed) back on a single machine. So the summation term becomes distributed.

Map Reduce can be applied wherever your learning algorithm can be expressed as a summation over your training set.

Map Reduce also works across multiple cores on a single computer.

9.30 Mean Shift Clustering

Mean shift clustering extends KDE one step further: the data points iteratively hill-climb to the peak of nearest KDE surface.

As a parameter to the kernel density estimates, you need to specify a *bandwidth* - this will affect the KDEs and their peaks, and thus it will affect the clustering results. You do not, however, need to specify the number of clusters.

Below are some examples of different bandwidth results ([source](#)).

You also need to make the choice of what kernel to use. Two commonly used kernels are:

- Flat kernel:

$$K(x) = \begin{cases} 1 & \text{if } \|x\| \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

- Gaussian kernel

Mean shift is slow ($\mathcal{O}(N^2)$).

9.30.1 References

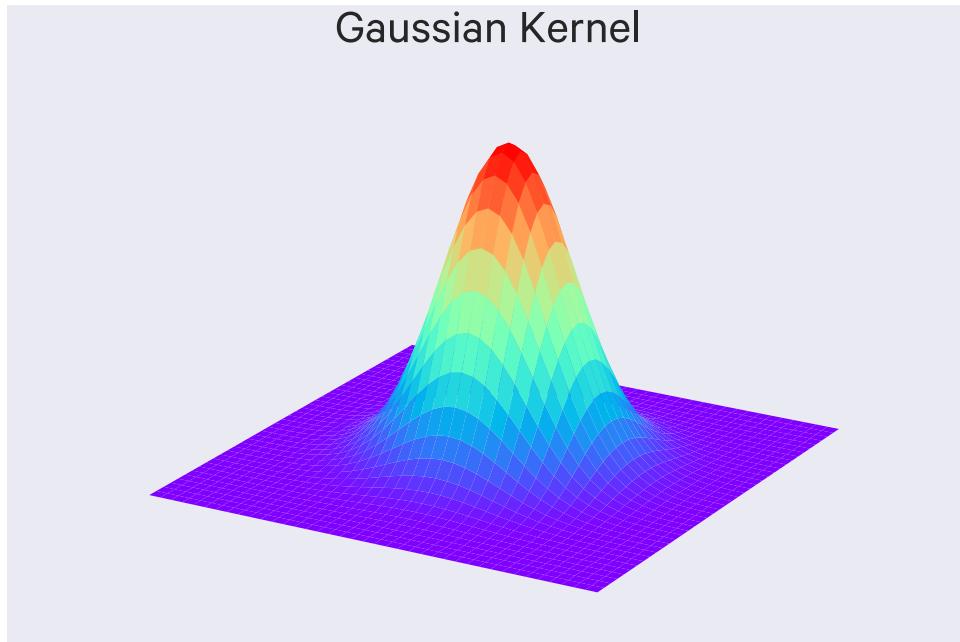
- [Mean Shift Clustering \(Matt Nedrich\)](#)

9.31 Predicting things

These are my notes are from Johns Hopkins' Practical Machine Learning course.

General approach:

- Start with a very specific and well-defined question: what do you want to predict, and what do you have to predict it with?



Gaussian kernel

- Try and find good input data
- Use features or features built from the data that may help with prediction
- Apply a machine learning algorithm
- Estimate the parameters for the algorithm
- Apply the algorithm to a data set
- Evaluate the results

9.31.1 Input data

John Tukey:

The combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.

The kind and quality of input data matters a lot - generally, if there is a direct relationship between your input data and what you're trying to predict (e.g. trying to predict new movie ratings based on old movie ratings), the task is easier.

Also, you have to ask - what is a “good prediction” in the context of your problem?

Generally, more data leads to better models.

9.31.2 Feature selection

Good features:

- lead to data compression
- retain relevant information
- are based on expert domain knowledge

Common mistakes:

- trying to automate feature selection
- not paying attention to data-specific quirks
- throwing away information unnecessarily

9.31.3 Algorithm selection

Algorithm selection is less important than you'd think - there may be a "best" algorithm for a particular problem, but often its performance is not much better than other well-performing approaches for that problem.

There may be certain qualities you look for in an ML algorithm:

- Interpretable - can we see or understand why the algorithm is making the decisions it makes?
- Simple - easy to explain and understand
- Accurate
- Fast (to train and test)
- Scalable (it can be applied to a large dataset)

Though there are generally trade-offs amongst these qualities.

9.31.4 In sample vs out of sample error

In sample error (aka *reubstitution error*): the error rate you get on the same data set you used to build your predictor.

Out of sample error (aka *generalization error*): The error rate you get on a new data set.

Out of sample error is more important; in sample error is always less than out of sample error so it is a much more optimistic result.

You don't want to overfit your predictor to the training data.

9.31.5 Prediction study design

- Randomly split data into:
 - training sample
 - testing sample
 - if enough data, a validation sample too
- Pick features on the training set
 - Use cross-validation
- Pick a prediction function on the training set
 - Use cross-validation
- If no validation data, apply the best prediction function to the test set *once*
- If you do have validation data
 - Apply the prediction function to the test set and refine
 - Apply the best prediction function to the validation set *once*

The key point is that you leave out a test set and treat it like new, unseen data so you can get a more honest evaluation of how your approach is performing.

9.31.6 Key quantities

For binary classification

- Sensitivity: $\frac{TP}{TP+FN}$
- Specificity: $\frac{TN}{TN+FP}$
- Positive predictive value: $\frac{TP}{TP+FP}$
- Negative predictive value: $\frac{TN}{TN+FN}$
- Accuracy: $\frac{TP+TN}{TP+FP+TN+FN}$

For continuous data

For evaluation metrics for continuous data (i.e. regression), you can look at *mean squared error* (MSE):

$$\frac{1}{n} \sum_{i=1}^n (\text{Prediction}_i - \text{Truth}_i)^2$$

Or the root mean squared error (RMSE):

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (\text{Prediction}_i - \text{Truth}_i)^2}$$

The MSE and RMSE are not robust to outliers, which is to say if the regression does exceptionally poorly on just one example, it can provide a misleading evaluation.

An alternative is to look at the **median absolute percentage**:

$$\text{MAPE} = \text{median}\left(\left|\frac{y_i - \hat{y}_i}{y_i}\right|\right)$$

9.31.7 Receiver Operating Characteristic (ROC) curves

In binary classification you may choose some cutoff above which you assign a sample to one class, and below which you assign a sample to the other class.

Depending on your cutoff, you will get different results - there is a trade off between the true and false positive rates.

You can plot a ROC curve, which has for its y-axis $P(TP)$ and for its x-axis $P(FP)$. Every point on the curve corresponds to a cutoff value. That is, the ROC curve visualizes a sweep through all the cutoff thresholds so you can see the performance of your classifier across *all* cutoff thresholds, whereas other metrics (such as the F-score and so on) only tell you the performance for one particular cutoff. By looking at all thresholds at once, you get a more complete and honest picture of how your classifier is performing, in particular, how well it is separating the classes. It is insensitive to the bias of the data's classes - that is, if there are way more or way less of the positive class than there are of the negative class (other metrics may be deceptively favorable or punishing in such unbalanced circumstances).

The *area under the curve* (AUC) is used to quantify how good the classification algorithm is. In general, an AUC of above 0.8 is considered “good”. An AUC of 0.5 (a straight line) is equivalent to random guessing.

So ROC curves (and the associated AUC metric) are very useful for evaluating binary classification.

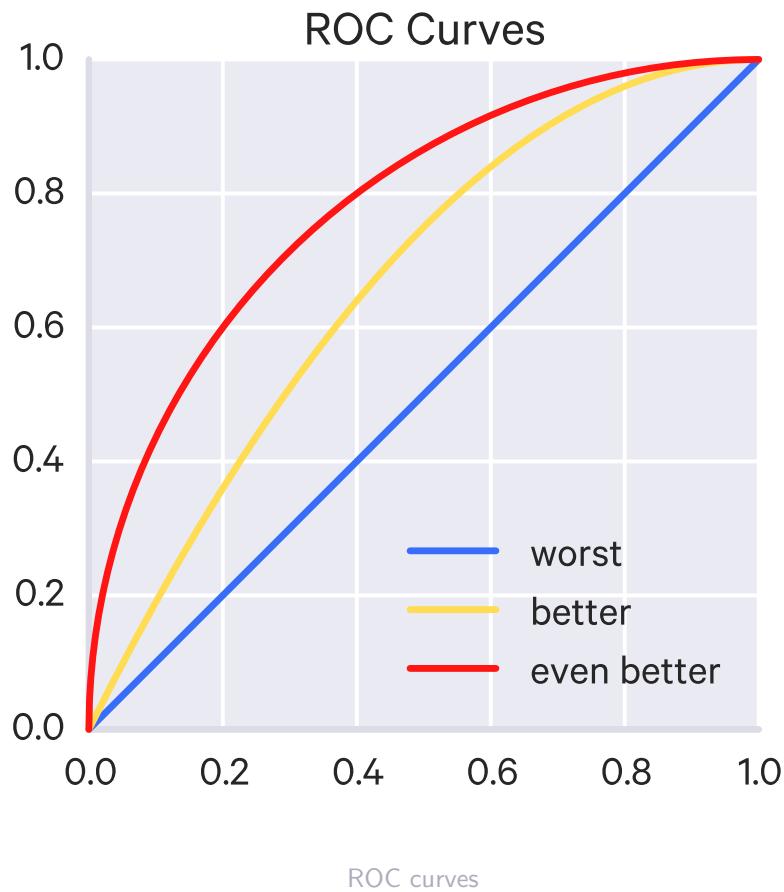
Note that ROC curves can be extended to classification of three or more classes by using the one-vs-all approach (see section on classification).

9.31.8 Cross validation

Cross-validation is a technique in which you take your training set (i.e. not your entire data - you still have your other test set which you are leaving aside) and split into a new training set and test set (perhaps better to call them “subsets”). You build a model on the training subset and evaluate on the test subset. You repeat this several times, re-splitting the training set into new training and testing subsets each time, then average the errors.

There are a few ways to build these subsets:

- *Random subsampling*: just assign randomly to the testing or training subset



- Not so good for time series data since you need continuous chunks
- Must be done without replacement, that is when you take a sample you must remove it so that you don't pick it twice in the same testing subset
- If you do it with replacement, it is called the *bootstrap*, which underestimates the error
- *K-fold*: split into k equal-sized subsets, e.g. take the first 1/3 of the training set as testing data, then take the second 1/3, then take the last 1/3.
 - Larger $k \Rightarrow$ less bias, more variance
 - Smaller $k \Rightarrow$ more bias, less variance
- *Leave one out*: just leave out one sample as the test subset, then take only the next sample as the test subset, then only the next, and so on, i.e. you iterate through all your samples as test subsets of size one.

9.31.9 Decision Trees

Basic algorithm:

1. Start with data all in one group
2. Find some criteria which best splits the outcomes
3. Divide the data into two groups (which become the leaves) on that split (which becomes a node)

4. Within each split, repeat
5. Repeat until the groups are too small or are sufficiently “pure” (homogeneous)

Classification trees are non-linear models:

- They use interactions b/w variables
- Data transformations may be less important (monotone transformations probably won't affect how data is split)
- Trees can be used for regression problems (continuous outcome)

Measures of impurity:

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in \text{Leaf}_m} \mathbb{1}(y_i = k)$$

That is, within the m leaf you have N_m objects to consider and you count the number of a particular class k in that set of objects and divide it by N_m to get the probability \hat{p}_{mk} .

Misclassification Error

$$1 - \hat{p}_{mk(m)}; k(m) = \text{most; common; } k$$

- 0 = perfect purity
- 0.5 = no purity

Gini index

$$\sum_{k \neq k'} \hat{p}_{mk} \times \hat{p}_{mk'} = \sum_{k=1}^K \hat{p}_{mk} (1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^K p_{mk}^2$$

- 0 = perfect purity
- 0.5 = no purity

Deviance/information gain:

$$-\sum_{k=1}^K \hat{p}_{mk} \log_2 \hat{p}_{mk}$$

- 0 = perfect purity
- 1 = no purity

9.31.10 Bagging (“Bootstrap aggregating”)

Sometimes if you average models together you get a better result.

Basic idea:

1. Resample cases and recalculate predictions
2. Average or majority vote

9.31.11 Random forests

Basic idea:

1. Bootstrap samples (i.e. resample)
2. At each split in the tree, bootstrap the variables (i.e. only a subset of the variables is considered at each split)
3. Grow multiple trees
4. Each tree votes on a classification

This can be very accurate but slow, prone to overfitting (cross-validation helps though), and not easy to interpret. However, they generally perform very well.

9.31.12 Boosting

Basic idea:

1. Take lots of (possibly) weak predictors h_1, \dots, h_k , e.g. a bunch of different trees or regression models or different cutoffs.
2. Weight them and combine them by creating a classifier which combines the predictors: $f(x) = \text{sign}(\sum_{t=1}^T \alpha_t h_t(x))$
 - Goal is to minimize error on training set
 - Iteratively select a classifier h at each step
 - Calculate weights based on errors
 - Increase the weight of missed classifications and select the next classifier
 - The sign of the result tells you the class

Adaboost is a popular boosting algorithm.

One class of boosting is *gradient boosting*.

Boosting typically does very well.

more on boosting

Here we focus on binary classification.

Say we have a classifier h which produces $+1$ or -1 .

We have some error rate, which ranges from 0 to 1. A weak classifier is one where the error is just less than 0.5 (that is, it works slightly better than chance). A stronger classifier has an error rate closer to 0.

Let's say we have several weak classifiers, h_1, \dots, h_n .

We can combine them into a bigger classifier, $H(x)$, where x is some input, which is the sum of the individual weak classifiers, and take the sign of the result. In this sense, the weak classifiers *vote* on the classification:

$$H(x) = \text{sign}(\sum_i h_i(x))$$

How do we generate these weak classifiers?

- We can create one by taking the data, training classifiers on it, and selecting with the smallest error rate (this will be classifier h_1 .)
- We can create another by taking the data and giving it some exaggeration of h_1 's errors (e.g. pay more attention to the samples that h_1 has trouble one). Training a new classifier on this gives us h_2 .
- We can create another by taking the data and giving it some exaggeration to the samples where the results of $h_1 \neq h_2$. Training a new classifier on this gives us h_3 .

This process can be recursive. That is, h_1 could be made up of three individual classifiers as well, and so could h_2 and h_3 .

For our classifiers we could use *decision tree stumps*, which is just a single test to divide the data into groups (i.e. just a part of a fuller decision tree). Note that boosting doesn't have to use decision tree (stumps), it can be used with any classifier.

We can assign a weight to each training example, w_i , where to start, all weights are uniform. These weights can be adjusted to exaggerate certain examples. For convenience, we keep it so that all weights sum to 1, $\sum w_i = 1$, thus enforcing a distribution.

We can compute the error ϵ of a given classifier as the sum of the weights of the examples it got wrong.

For our aggregate classifier, we may want to weight the classifiers with the weights $\alpha_1, \dots, \alpha_n$.

$$H(x) = \text{sign}(\sum_i \alpha_i h_i(x))$$

The general algorithm is:

- We can set the starting weights w_i^t for our training examples to be $\frac{1}{N}$ where N is the number of examples and $t = 1$, representing the time (or the iteration).
- Then we pick a classifier h^t which minimizes the error rate.
- Then we can pick α^t .
- And we can calculate w^{t+1} .
- Then repeat.

Now suppose $w_i^{t+1} = \frac{w_i^t}{Z} e^{-\alpha^t h^t(x) y(x)}$, where $y(x)$ gives you the right classification (the right sign) for a given Training example. So if $h^t(x)$ correctly classifies a sample, then it and $y(x)$ will be the same sign, so it will be a positive exponent. Otherwise, if $h^t(x)$ gives the incorrect sign, it will be a negative exponent. Z is some normalizing value so that we get a distribution.

We want to minimize the error bound for $H(x)$ if $\alpha^t = \frac{1}{2} / n \frac{1-\epsilon^t}{\epsilon^t}$.

9.32 Dirichlet Distribution

The Dirichlet distribution is a probability distribution over all possible multinomial distributions.

For example, say we have some data which we want to classify into three classes A, B, C . Maybe the data has 0.25 probability of being in class A , 0.5 probability of being in B , and 0.25 of being in C . Or maybe it has 0.1 probability of being in class A , then 0.6 and 0.3 for B and C respectively. Or it could be another distribution - we don't know. The Dirichlet distribution is the probability distribution representing these possible multinomial distributions across our classes.

The Dirichlet distribution is formalized as:

$$P(p|a) = \frac{\Gamma(\sum_{k=0}^{K-1} \alpha_k)}{\prod_{k=0}^{K-1} \Gamma(\alpha_k)} \prod_{k=0}^{K-1} p_k^{\alpha_k - 1}$$

where:

- p = a multinomial distribution
- α = the parameters of the dirichlet (a K -dimensional vector)
- K = the number of categories

Note that this term:

$$\frac{\Gamma(\sum_{k=0}^{K-1} \alpha_k)}{\prod_{k=0}^{K-1} \Gamma(\alpha_k)}$$

Is just a normalizing constant so that we get a distribution. So if you're just comparing ratios of these distributions you can ignore it.

You begin with some prior which can be derived from other data or from domain knowledge or intuition.

As more data comes in, we update the dirichlet (i.e. with Bayesian updates):

$$P(p|\text{data}) = \frac{P(\text{data}|p)P(p|\alpha)}{P(p)}$$

This can be done simply as updating the column in α which corresponds to a new data point, e.g. if we have three classes and $\alpha = [2, 4, 1]$ and we encounter a new data point which belongs to the class α_1 , we just add one to that column in α , so it becomes $[2, 5, 1]$.

9.32.1 Entropy

Also known as *information content*, *energy*, *log likelihood*, or $-\ln(p)$

It can be thought of as the amount of “surprise” for an event.

If an event is totally certain, it has zero entropy.

A coin flip has some entropy since there are only two equally-probable possibilities.

If you have a pair of dice, there is some entropy for rolling a 6 (because there are multiple combinations which can lead to 6) but much higher entropy for rolling a 12 (because there is only one combination which leads to a 12).

We can look at the entropy of the Dirichlet function:

$$\begin{aligned} E(p|\alpha) &= -\ln\left(\prod_{k_0}^{K-1} p_k^{\alpha_k-1}\right) \\ &= \sum_{k_0}^{K-1} (\alpha_k - 1)(-\ln(p_k)) \end{aligned}$$

We'll break out the entropy of a given multinomial distribution p into its own term:

$$e_k = -\ln(p_k)$$

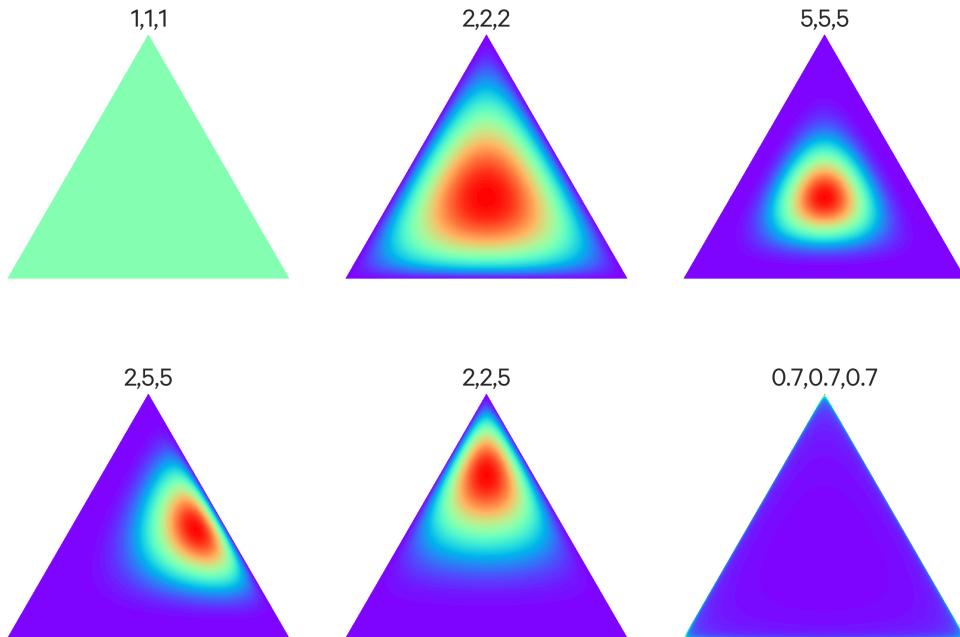
9.32.2 Interpreting α

We can take the α vector and normalize it. The normalized α vector is the expected value of the dirichlet, that is, it is its mean.

The sum of the unnormalized α vector is the weight of the distribution, which can be thought of as its precision. In a normal distribution, the precision is $\frac{1}{\text{variance}}$; a higher precision means a narrower normal distribution which means that values are likely to be near the mean. A lower precision means a wider distribution in which points are less likely to be near the mean.

So a dirichlet with a higher weight means that the multinomial distribution is more likely to be close to the expected value.

Dirichlet distributions can be thought of as a *simplex*, which is a generalization of a triangle in some arbitrary dimensions (e.g. in 2D it is 2-simplex, a triangle, in 3D it is 3-simplex, a pyramid, etc.). Some examples are below with their corresponding α vectors:



Some Dirichlet simplexes with their α vectors

9.33 Forecasting and timeseries prediction

Recommended reading: Rob Hyndman's *Forecasting: principles and practice*.

9.34 Maximum Likelihood Estimation (MLE)

Say we have some observed values x_1, x_2, \dots, x_n , generated by some latent model parameterized by θ , i.e. $f(x_1, x_2, \dots, x_n; \theta)$, where θ represents a single unknown parameter or a vector of unknown parameters. If we flip this we get the **likelihood** of θ , $L(\theta; x_1, x_2, \dots, x_n)$, which is the probability of θ , given the observed data.

The **maximum likelihood estimation** is the value of θ which maximizes this likelihood. That is, the value of θ which generates the observed values with the highest probability.

If the random variables associated with the values, i.e. X_1, X_2, \dots, X_n , are iid, then the likelihood is just:

$$L(\theta; x_1, x_2, \dots, x_n) = \prod_{i=1}^n f(x_i|\theta)$$

Sometimes this is just notated $L(\theta)$.

So we are looking to estimate the θ which maximizes this likelihood (this estimate is often notated $\hat{\theta}$, the hat typically indicates an estimator):

$$\hat{\theta} = \operatorname{argmax}_{\theta} L(\theta; x_1, x_2, \dots, x_n)$$

Logarithms are used, however, for convenience (i.e. dealing with sums rather than products), so instead we are often maximizing the **log likelihood** (which has its maximum at the same value (i.e. the same argmax) as the regular likelihood, though the actual maximum value may be different):

$$\ell(\theta) = \sum_{i=1}^n \log(f(x_i|\theta))$$

Example

Say we have a coin which may be unfair. We flip it ten times and get HHHHTTTTTT (we'll call this observed data X). We are interested in the probability of heads, π , for this coin, so we can determine if it's unfair or not.

Here we just have a binomial distribution so the parameters here are n and p (or π as we are referring to it here). We know n as it is the sample size, so that parameter is easy to "estimate" (i.e. we already know it). All that's left is the parameter p to estimate. So we can just use MLE to make this estimation; for binomial distributions it is rather trivial. Because p is the probability of a successful trial, and it's intuitive that the most likely p just reflects the number of observed successes over the total number of observed trials:

$$\begin{aligned}\tilde{\pi}_{MLE} &= \operatorname{argmax}_{\pi} P(X|\pi) \\ P(y|X) &\approx P(y|\tilde{\pi}_{MLE})\end{aligned}$$

Where y is the outcome of the next coin flip.

For this case our MLE would be $\tilde{\pi}_{MLE} = 0.4$ because that is most likely to have generated our observed data (we saw $\frac{4}{10}$ heads).

Also formulated as:

$$\hat{\theta} = \operatorname{argmax}_{\theta} p(x^{(1)}, \dots, x^{(n)})$$

For a Gaussian distribution, the sample mean is the MLE.

9.34.1 References

- Gibbs Sampling for the Uninitiated. Philip Resnik, Eric Hardisty. June 2010. <https://www.umiacs.umd.edu/~resnik/pubs/LAMP-TR-153.pdf>
- <https://onlinecourses.science.psu.edu/stat414/node/191>

9.35 Expectation Maximization

The **expectation maximization** (EM) algorithm is a two-staged iterative algorithm.

Say you have a dataset which is missing some values. How can you complete your data?

The EM algorithm allows you to do so.

The two stages work as such:

1. Begin with initial parameters $\hat{\theta}^{(t)}$, $t = 0$.
2. The “E-step”
 1. Using the current parameters $\hat{\theta}^{(t)}$, compute probabilities for each possible completion of the missing data.
 2. Use these probabilities to create a weighted training set of these possible completions.
3. The “M-step”
 1. Use a modified version of MLE (one which can deal with weighted samples) to derive new parameter estimates, $\hat{\theta}^{(t+1)}$.
4. Repeat the E and M steps until convergence.

Intuitively, what EM does is tries to find the parameters $\hat{\theta}$ which maximizes the log probability $\log P(x|\theta)$ of the observed data x , much like MLE, except does so under the conditions of incomplete data. EM will converge on a local optimum (maximum) for this log probability.

9.35.1 Example

Say we have two coins A, B , which may not be fair coins.

We conduct 5 experiments in which we randomly choose one of the coins (with equal probability) and flip it 10 times.

We have the following results:

1. HTTTHHTHTH
2. HHHHTHHHHH
3. HTHHHHHTHH
4. HTHTTTHHTT
5. THHHTHHHTH

We are still interested in learning a parameter for each coin, $\hat{\theta}_A, \hat{\theta}_B$, describing the probability of heads for each.

If we knew which coin we flipped during each experiment, this would be a simple MLE problem. Say we did know which coin was picked for each experiment:

1. B: HTTTHHTHTH
2. A: HHHHTHHHHH
3. A: HTHHHHHTHH
4. B: HTHTTTHHTT
5. A: THHHTHHHHTH

Then we just use MLE and get:

$$\begin{aligned}\hat{\theta}_A &= \frac{24}{24+6} = 0.8 \\ \hat{\theta}_B &= \frac{9}{9+11} = 0.45\end{aligned}$$

That is, for each coin we just compute $\frac{\text{num heads}}{\text{total trials}}$.

But, alas, we are missing the data of which coin we picked for each experiment. We can instead apply the EM algorithm.

Say we initially guess that $\hat{\theta}_A^{(0)} = 0.60, \hat{\theta}_B^{(0)} = 0.50$. For each experiment, we'll compute the probability that coin A produced those results and the same probability for coin B . Here we'll just show the computation for the first experiment.

We're dealing with a binomial distribution here, so we are using:

$$P(x) = \binom{n}{x} p(1-p)^{n-x}, \quad p = \hat{\theta}$$

The binomial coefficient is the same for both coins (the $\binom{n}{x}$ term) and cancels out in normalization, so we only care about the remaining factors. So we will instead just use:

$$P(x) = p(1-p)^{n-x}, \quad p = \hat{\theta}$$

For the first experiment we have 5 heads (and $n = 10$). Using our current estimates for $\hat{\theta}_A, \hat{\theta}_B$, we compute:

$$\begin{aligned}\theta_A^5(1-\theta_A)^{10-5} &\approx 0.0008 \\ \theta_B^5(1-\theta_B)^{10-5} &\approx 0.0010 \\ \frac{0.0008}{0.0008 + 0.0010} &\approx 0.44 \\ \frac{0.0010}{0.0008 + 0.0010} &\approx 0.56\end{aligned}$$

So for this first iteration and for the first experiment, we estimate that the chance of the picked coin being coin A is about 0.44, and about 0.56 for coin B .

Then we generate the weighted set of these possible completions by computing how much each of these coins, as weighted by the probabilities we just computed, contributed to the results for this experiment ((5H, 5T)):

$$\begin{aligned} 0.44(5H, 5T) &= (2.2H, 2.2T), \text{ (coin A)} \\ 0.56(5H, 5T) &= (2.8H, 2.8T), \text{ (coin B)} \end{aligned}$$

Then we repeat this for the rest of the experiments, getting the following weighted values for each coin for each experiment:

| coin A | coin B |
|------------|------------|
| 2.2H, 2.2T | 2.8H, 2.8T |
| 7.2H, 0.8T | 1.8H, 0.2T |
| 5.9H, 1.5T | 2.1H, 0.5T |
| 1.4H, 2.1T | 2.6H, 3.9T |
| 4.5H, 1.9T | 2.5H, 1.1T |

and sum up the weighted values for each coin:

| coin A | coin B |
|-------------|-------------|
| 21.3H, 8.6T | 11.7H, 8.4T |

Then we use these weighted values and MLE to update $\hat{\theta}_A, \hat{\theta}_B$, i.e.:

$$\begin{aligned} \hat{\theta}_A^{(1)} &\approx \frac{21.3}{21.3 + 8.6} \approx 0.71 \\ \hat{\theta}_B^{(1)} &\approx \frac{11.7}{11.7 + 8.4} \approx 0.58 \end{aligned}$$

And repeat until convergence.

9.35.2 Expectation Maximization as a Generalization of K-Means

In K-Means we make hard assignments of datapoints to clusters (that is, they belong to only one cluster at a time, and that assignment is binary).

EM is similar to K-Means, but we use soft assignments instead - datapoints can belong to multiple clusters in varying strengths. When the centroids are updated, they are updated against *all* points, weighted by assignment strength (whereas in K-Means, centroids are updated only against their members).

EM converges to approximately the same clusters as K-Means, except datapoints still have some membership to other clusters (though they may be very weak memberships).

In EM, we consider that each datapoint is generated from a mixture of classes.

For each K classes, we have the prior probability of that class $P(C = i)$ and the probability of the datapoint given that class $P(x|C = i)$.

$$P(x) = \sum_{i=1}^K P(C = i)P(x|C = i)$$

These terms may be notated:

$$\begin{aligned}\pi &= P(C = i) \\ \mu_i \sum_i &= P(x|C = i)\end{aligned}$$

What this is modeling here is that each centroid is the center of a Gaussian distribution, and we try to fit these centroids and their distributions to the data.

9.35.3 References

- <http://www.nature.com/nbt/journal/v26/n8/full/nbt1406.html>
- <https://math.stackexchange.com/questions/25111/how-does-expectation-maximization-work>
- <https://math.stackexchange.com/questions/81004/how-does-expectation-maximization-work-in-coin-flip>
- Bayesian Inference with Tears. Kevin Knight, September 2009.
- Introduction to Artificial Intelligence (Udacity CS271): <https://www.udacity.com/wiki/cs271>, Peter Norvig and Sebastian Thrun.

9.36 Maximum a posteriori (MAP) estimation

Alternative to MLE, we can estimate probabilities using *maximum a priori estimation*, where we instead choose a probability that is most likely given the observed data:

$$\begin{aligned}\tilde{\pi}_{MAP} &= \operatorname{argmax}_\pi P(\pi|X) \\ &= \operatorname{argmax}_\pi \frac{P(X|\pi)P(\pi)}{P(X)} \\ &= \operatorname{argmax}_\pi P(X|\pi)P(\pi) \\ P(y|X) &\approx P(y|\tilde{\pi}_{MAP})\end{aligned}$$

So unlike MLE, MAP estimation uses Bayes' Rule so the estimate can use prior knowledge ($P(\pi)$) about what we expect π to be.

9.36.1 References

- Gibbs Sampling for the Uninitiated. Philip Resnik, Eric Hardisty. June 2010. <https://www.umiacs.umd.edu/~resnik/pubs/LAMP-TR-153.pdf>

9.37 Markov Chain Monte Carlo (MCMC)

9.37.1 Motivation

With MLE and MAP estimation we get only a single value for π , and this collapsing into a single value loses information - what if we instead considered the entire distribution of values for π , i.e. $P(\pi|X)$?

As it stands, with MLE and MAP we only get an approximation of $P(y|X)$. But with the distribution $P(\pi|X)$ we could directly compute its expected value:

$$E[P(y|X)] = \int P(y|\pi)P(\pi|X)d\pi$$

And with Bayes' Rule we have:

$$P(\pi|X) = \frac{P(X|\pi)P(\pi)}{P(X)} = \frac{P(X|\pi)P(\pi)}{\int_{\pi} P(X|\pi)P(\pi)d\pi}$$

So we have two integrals here, and unfortunately integrals can be hard (sometimes impossible) to compute.

With MCMC we can get the values we need without needing to calculating the integrals.

9.37.2 Monte Carlo methods

Monte Carlo methods are algorithms which perform probabilistic simulations to give you some value.

For example:

Say you have a square and a circle inscribed within it, so that they are co-centric and the circle's diameter is equal to the length of a side of the square. You take some rice and uniformly scatter it in the shapes at random. You can count the total number of grains of rice in the circle (C) and do the same for rice in the square (S). The ratio $\frac{C}{S}$ approximates the ratio of the area of the circle to the area of the square. The area of the circle and for the square can be thought of as integrals (adding an infinite number of infinitesimally small points), so what you have effectively done is approximate the value of integrals.

9.37.3 MCMC

In the example the samples were uniformly distributed, but in practice they can be drawn from other distributions. If we collect enough samples from the distribution we can compute pretty much anything we would want to know about the distribution - mean, standard deviation, etc.

For example, we can compute the expected value:

$$E[f(z)] = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{t=1}^N f(z^{(t)})$$

Since we don't sample infinite points, we sample as many as we can for an approximation:

$$E[f(z)] \approx \frac{1}{T} \sum_{t=1}^T f(z^{(t)})$$

How exactly then is the sampling of $z^{(0)}, \dots, z^{(T)}$ according to a given distribution accomplished?

We treat the sampling process as a walk around a sample space and the walk proceeds as a Markov chain; that is, the choice of the next sample depends on only the current state, based on a transition probability $P_{\text{trans}}(z^{(t+1)}|z^{(t)})$.

So the general walking algorithm is:

- Randomly initialize $z^{(0)}$
- for $t = 1$ to T do:
 - $z^{(t+1)} := g(z^{(t)})$

Where g is just a function which returns the next sample based on P_{trans} and the current sample.

9.37.4 Gibbs Sampling

Gibbs sampling is an MCMC algorithm, where z is a point/vector $[z_1, \dots, z_k]$ and $k > 1$. So here the samples are vectors of at least two terms. You don't select an entire sample at once, what you do is make a separate probabilistic choice for each dimension, where the choice is dependent on the other $k - 1$ dimensions, using the *newest* values for each.

For example, say $k = 3$ so you have vectors in the form $[z_1, z_2, z_3]$.

- First you pick a new value $z_1^{(t+1)}$ based on $z_2^{(t)}$ and $z_3^{(t)}$.
- Then you pick a new value $z_2^{(t+1)}$ based on $z_1^{(t+1)}$ and $z_3^{(t)}$.
- Then you pick a new value $z_3^{(t+1)}$ based on $z_1^{(t+1)}$ and $z_2^{(t+1)}$.

9.37.5 References

- Gibbs Sampling for the Uninitiated. Philip Resnik, Eric Hardisty. June 2010. <https://www.umiacs.umd.edu/~resnik/pubs/LAMP-TR-153.pdf>

9.38 Softmax regression

Softmax regression generalizes logistic regression to beyond binary classification (i.e. multinomial classification; that is, there are more than just two possible classes). Logistic regression is the reduced form of softmax regression where $k = 2$ (thus logistic regression is sometimes called a “binary Softmax classifier”). As is with logistic regression, softmax regression outputs probabilities for each class. As a generalization of logistic regression, softmax regression can also be expressed as a generalized linear model. It generally uses a cross-entropy loss function.

9.38.1 Hierarchical Softmax

In the case of many, many classes, the *hierarchical* variant of Softmax may be preferred. In hierarchical Softmax, the labels are structured as a hierarchy (a tree). A Softmax classifier is trained for each node of the tree, to distinguish the left and right branches.

9.39 Factor Analysis

Factor analysis uses principal component analysis (PCA) to reduce dimensionality of data.

It is based on an orthogonal decomposition of an input matrix to yield an output matrix of orthogonal components (factors) that maximize the amount of variation in the variables from the input matrix.

In factor analysis, the retained principal components are called *common factors* and their correlations with the input variables are called *factor loadings*.

PCA becomes more reliable the more data you have. The number of examples must be larger than the number of variables in the input matrix. The assumptions of linear correlation must hold as well (i.e. that the variables must be linearly related).

9.40 Discriminant Function Analysis (DFA)

The goal here is to maximize the distance between two or more groups, which in turn maximizes *discriminability* through a set of one or more functions of a specific rank, i.e. the number of functions required to maximize the separation between groups. These functions are typically linear combinations of the input variables and are called *linear discriminant functions* (LDFs).

9.41 Non-Negative Matrix Factorization (NMF)

9.41.1 Matrix factorization

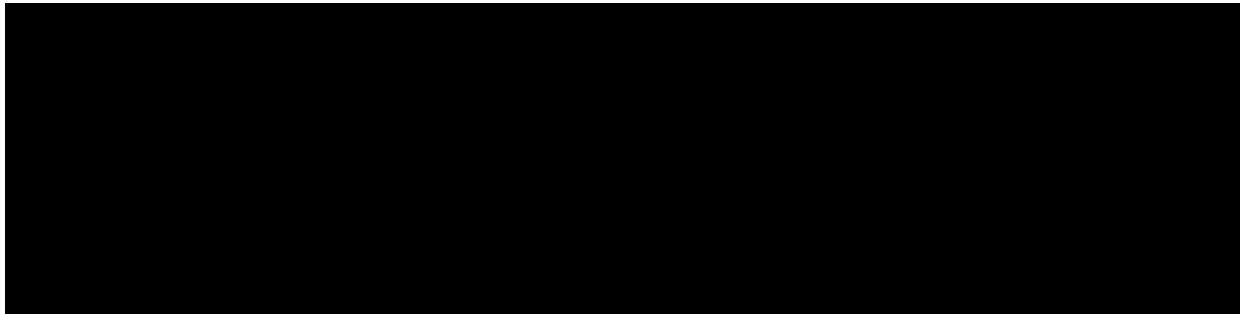
Let V be an $m \times n$ matrix of rank r . Then there is an $m \times r$ matrix W and an $r \times n$ matrix H such that $V = WH$. So we can factorize (or *decompose*) V into W and H .

This matrix factorization can be seen as a form of compression (for low rank matrices, at least) - if we were to store V on its own, we have to store $m \times n$ elements, but if we store W and H separately, we only need to store $m \times r + r \times n$ elements, which will be smaller than $m \times n$ for low rank matrices.

Note that this kind of factorization can't be solved analytically, so it is usually approximated numerically (there are a variety of algorithms for doing so).

9.41.2 Non-Negative Matrix Factorization (NMF)

NMF is a particular matrix factorization in which each element of V is ≥ 0 (a non-negative constraint), and results in factor matrices W and H such that each of their elements are also ≥ 0 .



Non-negative matrix factorization (By Qwertys, CC BY-SA 3.0, via Wikimedia Commons)

Each column v_i in V can be calculated from W and H like so (where h_i is a column in H):

$$v_i = Wh_i$$

NMF is useful for reducing (i.e. compressing) the dimensionality of a dataset, in particular, it reduces it into a linear combination of bases.

NMF can also be used for clustering; it has a consequence of naturally clustering the columns of V .

If you add an orthogonality constraint, i.e. $HH^T = I$, if the value at $H_{kj} > 0$, then the j th column of V , that is, v_j , belongs to the cluster k .

9.41.3 References

- <https://www.youtube.com/watch?v=o8PiWO8C3zs>
- https://en.wikipedia.org/wiki/Non-negative_matrix_factorization#Clustering_property

9.42 Spectral Clustering (Affinity-Based Clustering)

With spectral clustering, datapoints are clustered by *affinity* - that is, by nearby points - rather than by centroids (as is with K-Means). Using affinity instead of centroids, spectral clustering can identify clusters where K-Means fails to.

In spectral clustering, an *affinity matrix* is produced which, for a set of n datapoints, is an $n \times n$ matrix. Pairwise affinities are computed for the dataset. Affinity is some distance metric.

Then, from this affinity matrix, PCA is used to extract the eigenvectors with the largest eigenvalues and the data is then projected to the new space defined by PCA. The data will be more clearly separated in this new representation such that conventional clustering methods (e.g. K-Means) can be applied.

9.43 Information Gain & Entropy

One way of thinking about entropy is: With high entropy, it is very hard to guess the value of the random variable (because all values are equally or similarly likely); with low entropy it is easy to guess its value (because there are some values which are much more likely than the others).

The entropy of a random variable X is computed:

$$H(X) = -\sum_{j=1}^m p_j \log p_j$$

Where X can take on m possible values and p_j is the probability of X taking on the j th value.

9.43.1 Specific Conditional Entropy

The specific conditional entropy $H(Y|X = v)$ is the entropy of some random variable conditioned on another random variable taking some value.

9.43.2 Conditional Entropy

The conditional entropy $H(Y|X)$ is the entropy of some random variable conditioned on another random variable, i.e. it is the average specific conditional entropy of Y , that is:

$$\sum_j P(X = v) H(Y|X = v)$$

9.43.3 Information Gain

Say you must transmit the random variable Y . How many bits on average would be saved if both the sender and the recipient knew X ?

To put it more concretely:

$$IG(Y|X) = H(Y) - H(Y|X)$$

The bigger the difference, the more X tells us about Y (because it decreases the entropy, i.e. it makes it easier to guess Y).

9.44 CURE (Clustering Using Representatives)

If you are dealing with more data than can fit into memory, you may have issues clustering it.

A flexible clustering algorithm (there are no restrictions about the shape of the clusters it can find) which can handle massive datasets is CURE.

CURE uses Euclidean distance and generates a set of k representative points for each cluster. It uses these points to represent clusters, therefore avoiding the need to store every datapoint in memory.

CURE works in two passes.

For the first pass, a random sample of points from the dataset are chosen. The more samples the better, so ideally you choose as many samples as can fit into memory. Then you apply a conventional clustering algorithm, such as hierarchical clustering, to this sample. This creates an initial set of clusters to work with.

For each of these generated clusters, we pick k representative points, such that these points are as dispersed as possible within the cluster.

For example, say $k = 4$. For each cluster, pick a point at random, then pick the furthest point from that point (within the same cluster), then pick the furthest point (within the same cluster) from those two points, and repeat one more time to get the fourth representative point.

Then copy each representative point and move that copy some fixed fraction (e.g. 0.2) closer to the cluster's centroid. These copied points are called “synthetic points” (we use them so we don't actually move the datapoints themselves). These synthetic points are the representatives we end up using for each cluster.

For the second pass, we then iterate over each point p in the entire dataset. We assign p to its closest cluster, which is the cluster that has the closest representative point to p .

9.44.1 References

- *Mining Massive Datasets* (Coursera & Stanford, 2014). Jure Leskovec, Anand Rajaraman, Jeff Ullman.

9.45 Model Validation

Is the process of measuring how your model performs on new/unseen data. This is where you try to determine if your model over fits or if it generalizes well to other data.

9.46 Other Loss Functions

The squared error loss function is not the only loss function available. There are a variety you can use, and you can even come up with your own if needed. Perhaps, for instance, you want to weigh positive errors more than negative errors.

9.46.1 Hinge Loss (aka Max-Margin Loss)

The hinge loss function takes the form $\ell(y) = \max(0, 1 - t \cdot y)$ and is typically used for SVMs (sometimes squared hinge loss is used, which is just the previous equation squared). (TODO add more info)

9.46.2 Cross-entropy loss

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{n \in N} \sum_{i \in C} y_{n,i} \log \hat{y}_{n,i}$$

where

- N = number of samples
- C = number of classes

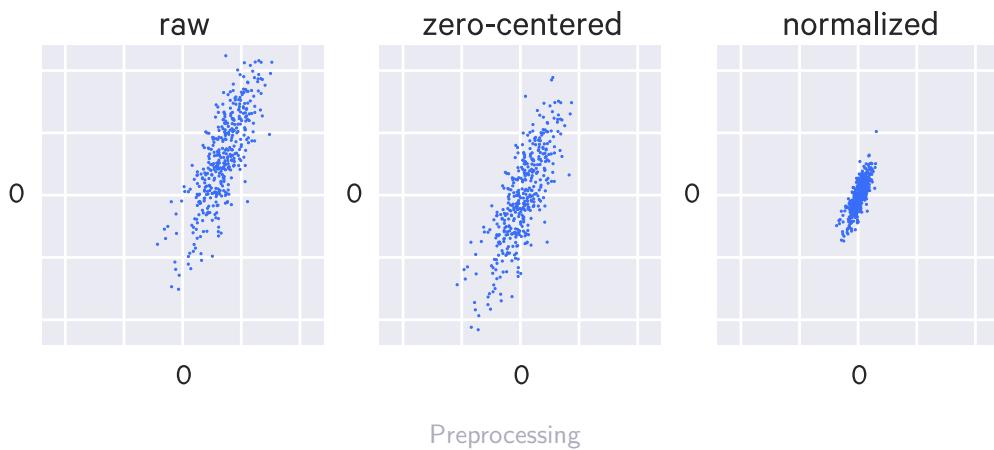
Typically used for Softmax classifiers.

$$L_i = -\log \left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right)$$

9.47 Data preprocessing

9.47.1 Mean subtraction

Mean subtraction centers the data around the origin (i.e. it “zero-centers” it), simply by subtracting each feature’s mean from itself.



9.47.2 Normalization

(written elsewhere)

9.47.3 References

- CS231n Convolutional Neural Networks for Visual Recognition, Module 1: Neural Networks Part 2: Setting up the Data and the Loss. Andrej Karpathy. <https://cs231n.github.io/neural-networks-1/>

9.48 Debugging Neural Networks/Choosing hyperparameters

TODO See: <https://cs231n.github.io/neural-networks-3/#anneal>

9.49 Generalized Linear Models

Linear regression is a useful linear model but requires that some assumptions hold, such as that the error is normally distributed. There may be times where regression can't fit a good line - in such cases, a different generalized linear model (GLM, of which one is the normal linear regression) may be more appropriate.

9.49.1 Logistic Regression

Logistic regression is also a GLM - you're fitting a line which models the probability of being in the positive class. We can use the Bernoulli distribution since it models events with two possible outcomes and is parameterized by only the probability of the positive outcome, p . Thus our line would look something like:

$$p_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + \epsilon$$

But to represent a probability, y values must be bound to $[0, 1]$. Currently, our model can be linear or polynomial and thus can output any continuous value. So we have to apply a transformation to constrain y ; we do so by applying a **logit transformation**:

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) = x$$

The $\frac{p}{1-p}$ term constraints the output to be positive. The log operation constrains the values to $[0, 1]$.

The inverse of the logit transformation is:

$$p = \frac{1}{1 + \exp(-x)}$$

So the model is now:

$$\text{logit}(p) = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + \epsilon$$

So the likelihood here is:

$$L(y|p) = \prod_{i=1}^n p_i^{y_i} (1 - p_i)^{1-y_i}$$

And the log likelihood then is:

$$l(y|p) = \sum_{i=1}^n y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

9.50 References

- IFT 725 Review of fundamentals
- [Exploratory Data Analysis Course Notes](#), Xing Su
- Johns Hopkins' Data Science Specialization (Coursera 2015)
- *Mining Massive Datasets* (Coursera & Stanford, 2014). Jure Leskovec, Anand Rajaraman, Jeff Ullman.
- Andrew Ng's Coursera *Machine Learning* course (2014)
- MIT 6.034 (Fall 2010): Artificial Intelligence. Patrick H. Winston.

- Computational Statistics II. Chris Fonnesbeck. SciPy 2015: <https://www.youtube.com/watch?v=heFaYLVZ4> and https://github.com/fonnesbeck/scipy2015_tutorial
 - Introduction to Artificial Intelligence (Udacity CS271): <https://www.udacity.com/wiki/cs271>, Peter Norvig and Sebastian Thrun.
-

9.51 Regression

Regression involves fitting a model to data. The goal is to understand the relationship between one set of variables - the **dependent** or **response** or **target** or **outcome** or **explained** variables (e.g. y) - and another set - the **independent** or **explanatory** or **predictor** or **regressor** variables (e.g. X or x). In cases of just one dependent and one explanatory variable, we have **simple regression**. In scenarios with more than one explanatory variable, we have **multiple regression**. In scenarios with more than one dependent variable, we have **multivariate regression**.

With **linear regression** we expect that the dependent and explanatory variables have a linear relationship; that is, can be expressed as a **linear combination** of random variables, i.e.:

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + \varepsilon$$

For some dependent variable y and explanatory variables x_1, \dots, x_n , where ε is the residual due to random variation or other noisy factors.

Of course, we do not know the true values for these β parameters (also called **regression coefficients**) so they end up being point estimates as well. We can estimate them as follows.

When given data, one technique we can use is **ordinary least squares**, sometimes just called **least squares regression**, which looks for parameters β_0, \dots, β_n such that the sum of the squared residuals (i.e. the SSE, i.e. $e_1^2 + \cdots + e_n^2 = \sum_{i=1}^n (y_i - \hat{y}_i)^2$) is minimized (this minimization requirement is called the **least squares criterion**). The resulting line is called the **least squares line**.

Note that *linear model* does not mean the model is necessarily a *straight* line. It can be polynomial as well - but you can think of the polynomial terms as additional explanatory variables; looking at it this way, the line (or curve, but for consistency, they are all called "lines") still follows the form above. And of course, in higher-dimensions (that is, for multiple regression) we are not dealing with lines but planes, hyperplanes, and so on. But again, for the sake of simplicity, they are all just referred to as "lines".

For example, the line:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \varepsilon$$

Can be re-written:

$$x_2 = x_1^2$$

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

When we use a regression model to predict a dependent variable, e.g. y , we denote it as a estimate by putting a hat over it, e.g. \hat{y} .

9.51.1 Evaluating model quality

We can measure our model's quality by measuring the variance explained, that is, with R^2 . Here we can use this formulation of it:

$$R^2 = 1 - \frac{\text{variability in residuals}}{\text{variability in the outcome}} = 1 - \frac{\text{Var}(e_i)}{\text{Var}(y_i)}$$

For multiple regression, we use a slightly different formulation, called **adjusted R^2** , because in multiple regression the regular R^2 is a biased estimate:

$$R_{\text{adj}}^2 = 1 - \frac{\text{Var}(e_i)}{\text{Var}(y_i)} \frac{n - 1}{n - k - 1}$$

Where n is the sample size used to fit the model and k is the number of predictor variables in the model.

A higher R^2 or adjusted R^2 indicates a better model fit.

Alternatively, you can measure model quality using p-values. Your null hypothesis is that the true linear model has slope zero, and your alternate hypothesis is that it has a non-zero slope (you can limit to less than 0 or greater than 0 depending on what your fit model is showing you; e.g. if your fit model has a positive slope, then your alternate hypothesis is that the true linear model has a positive slope).

So to test the hypothesis, you'd identify a standard error for the estimated parameters, compute the appropriate test statistic (usually a t value), and then identify the p-value.

For example, say your linear model has the slope -1. You could compute your t value T , once you have your standard error, like so:

$$T = \frac{\text{estimate} - \text{null value}}{\text{SE}} = \frac{-1 - 0}{\text{SE}}$$

9.51.2 Outliers

Outliers can pose a problem for fitting a regression line. Outliers that fall horizontally away from the rest of the data points can influence the line more, so they are called points with **high leverage**.

Any such point that actually does influence the line's slope is called an **influential point**. You can examine this effect by removing the point and then fitting the line again and seeing how it changes.

Outliers should only be removed with good reason - they can still be useful and informative and a good model will be able to capture them in some way.

9.51.3 Extrapolation

With linear models, you should avoid **extrapolation**, that is, estimating values which are outside the original data's range. For example, if you have data in some range $[x_1, x_n]$, you have no guarantee that your model behaves correctly at $x < x_1$ and $x > x_n$.

9.51.4 Collinearity

Sometimes in multiple regression you may have predictor variables which are correlated with one another; we say that these predictors are **collinear**.

9.51.5 Model Selection

When you have a set of different explanatory variables (i.e. multiple regression), you have to decide which ones to include and which ones to toss out. This process is an example of **model selection**.

The model where you include all available explanatory variables is called the **full model**. But sometimes including all explanatory variables can hurt prediction accuracy.

There are a few model selection strategies that are used.

One class of selection strategies is called **stepwise** model selection because they iteratively remove or add one explanatory variable at a time, measuring the goodness of fit for each. The two approaches here are the **backward-elimination** strategy which begins with the full model and removes one explanatory variable at a time, and the **forward-selection** strategy which is the reverse of backward-elimination, starting with one explanatory variable and adding the rest one at a time. These two strategies don't necessarily lead to the same model; if you use both, pick the one with the better adjusted R^2 score.

9.51.6 Logistic regression

Linear regression is good for explaining continuous dependent variables. But for discrete variables, linear regression gives ambiguous results - what does a fractional result mean? It can't be interpreted as a probability because linear regression models are not bound to $[0, 1]$ as probability functions must be.

When dealing with boolean/binary dependent variables you can use **logistic regression**. When dealing with non-binary discrete dependent variables, you can use **Poisson regression** (which is a GLM that uses the log link function).

So we expect the logistic regression function to output a probability. In linear regression, the model can output any value, not bound to $[0, 1]$. So for logistic regression we apply a transformation, most commonly the **logit transformation**, so that our resulting values can be interpreted as probability:

$$\text{transformation}(p) = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$$

$$\text{logit}(p) = \log_e\left(\frac{p}{1-p}\right)$$

So if we solve the original regression equation for p , we end up with:

$$p = \frac{e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n}}{1 + e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n}}$$

Logistic regression does not have a closed form solution - that is, it can't be solved in a finite number of operations, so we must estimate its parameters using other methods, more specifically, we use iterative methods. Generally the goal is to find the **maximum likelihood estimate** (MLE), which is the set of parameters that maximizes the likelihood of the data. So we might start with random guesses for the parameters, then compute the likelihood of our data (that is, we can compute the probability of each data point; the likelihood of the data is the product of these individual probabilities) based on these parameters. We iterate until we find the parameters which maximize this likelihood.

9.51.7 Generalized linear models (GLM)

We can use linear models for non-regression situations - that is, when the output variable is not an unbounded continuous value directly computed from the inputs (that is, the output variable is not a linear function of the inputs), such as with binary or other kinds of classification. In such cases, the linear models we used are called *generalized linear models*. Like any linear function, we get some value from our inputs, but we then also apply a *link function* which transforms the resulting value into something we can use. Another way of putting it is that these link functions allow us to *generalize* linear models to other situations.

Linear regression also assumes *homoscedasticity*; that is, that the variance of the error is uniform along the line. GLMs do not need to make this assumption; the link function transforms the data to satisfy this assumption.

For example, say you want to predict whether or not someone will buy something - this is a binary classification and we want either a 0 or a 1. We might come up with some linear function based on income and number of items purchased in the last month, but this won't give us a 0/no or a 1/yes, it will give us some continuous value. So then we apply some link function of our choosing which turns the resulting value to give us the probability of a 1/yes.

Linear regression is also a GLM, where the link function is the identity function.

Logistic regression uses the logit link function.

Logistic regression is a type of models called **generalized linear models** (GLM), which involves two steps:

1. Model the response variable with a probability distribution.
2. Model the distribution's parameters using the predictor variables and a special form of multiple regression.

This probability distribution is taken from the exponential family of probability distributions, which includes the normal, Bernoulli, beta, gamma, Dirichlet, and Poisson distributions (among others). A distribution is in the exponential family if it can be written in the form:

$$P(y|\eta) = b(y) \exp(\eta^T T(y) - a(\eta))$$

η is known as the **natural parameter** or the **canonical parameter** of the distribution, $T(y)$ is the **sufficient statistics**, which is often just $T(y) = y$. $a(\eta)$ is the **log partition function**.

We can set T, a, b to define a family of distributions; this family is parameterized by η , with different values giving different distributions within the family.

For instance, the Bernoulli distribution is in the exponential family, where

$$\begin{aligned}\eta &= \log\left(\frac{p}{1-p}\right) \\ T(y) &= y \\ a(\eta) &= -\log(1-p) \\ b(y) &= 1\end{aligned}$$

Same goes for the Gaussian distribution, where

$$\begin{aligned}\eta &= \mu \\ T(y) &= y \\ a(\eta) &= \frac{\mu^2}{2} \\ b(y) &= \frac{1}{\sqrt{(2\pi)}} \exp\left(-\frac{y^2}{2}\right)\end{aligned}$$

9.52 Linear Mixed Models (LMM), or just “Mixed Models” or “Hierarchical Linear Models”

In a linear model there may be *mixed effects*, which includes *fixed* and *random* effects. Fixed effects are variables in your model where their coefficients are fixed (non-random). Random effects are variables in your model where their coefficients are random.

For example, say you want to create a model for crop yields given a farm and amount of rainfall. We have data from several years and the same farms are represented multiple times throughout. We could

consider that some farms may be better at producing greater crop yields given the same amount of rainfall as another farm. So we expect that samples from different farms will have different variances - e.g. if we look at just farm A's crop yields, that sample would have different variance than if we just looked at farm B's crop yields. In this regard, we might expect that models for farm A and farm B will be somewhat different.

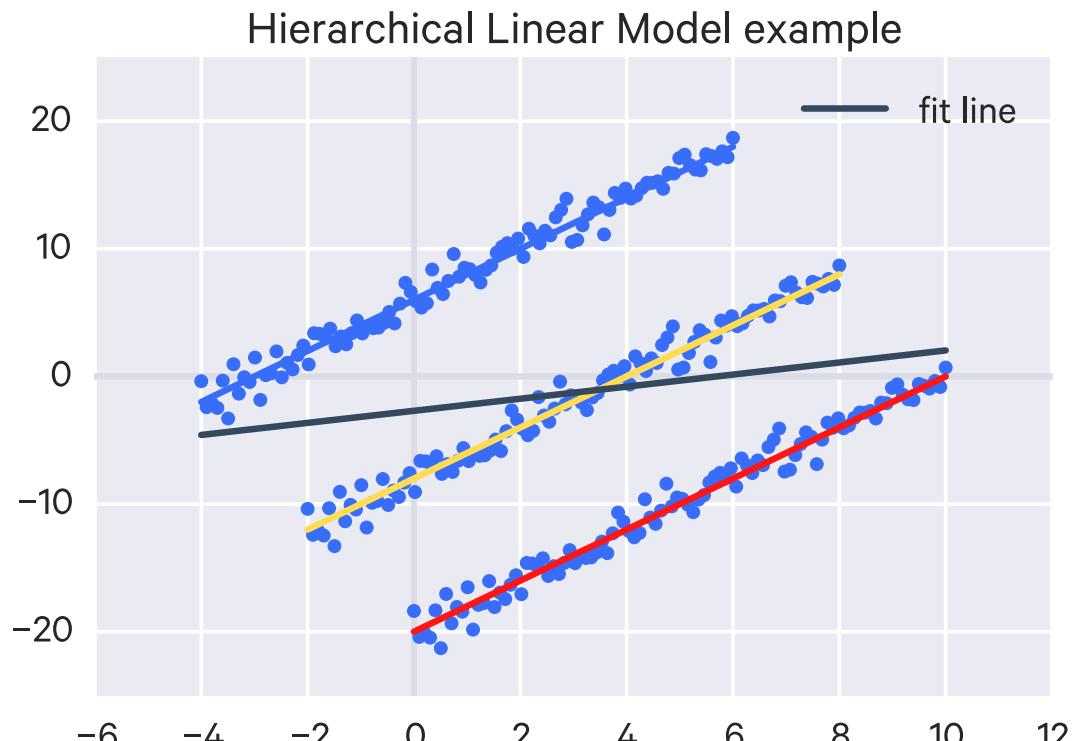
The naive approach would be to just ignore differences between farms and consider only rainfall as a fixed effect (i.e. with a fixed/constant coefficient). This is sometimes called “pooling” because we've lumped everything (in our case, all the farms) together.

We could create individual models for each farm (“no pooling”) but perhaps for some farms we only have one or two samples. For those farms, we'd be building very dubious models since their sample sizes are so small. The information from the other farms are still useful for giving us more data to work with in these cases, so no pooling isn't necessarily a good approach either.

We can use a mixed model (“partial pooling”) to capture this and make it so that the rainfall coefficient random, varying by farm.

more...from another source

We may run into situations like the following:

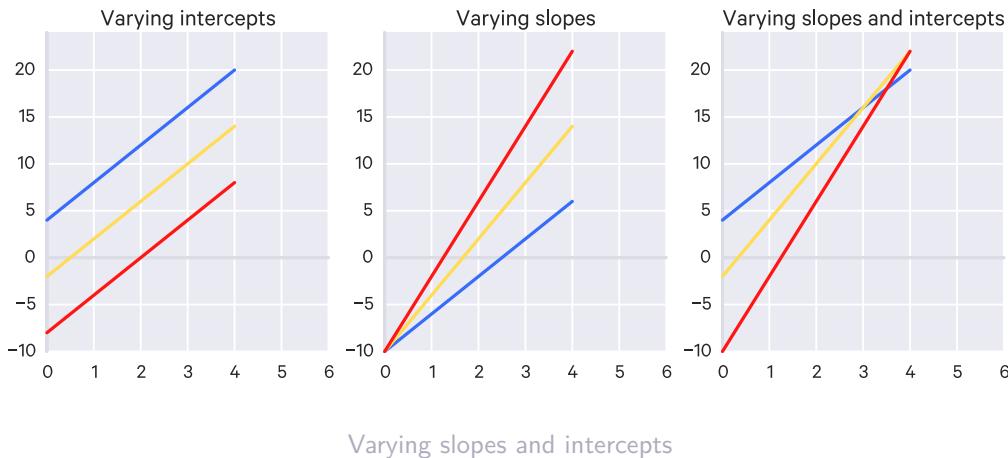


A situation where an HLM might be better

Where our data seems to encompass multiple models (the red, green, blue, and black ones going up from left to right), but if we try to model them all simultaneously, we get a complete incorrectly model (the dark grey line going down from left to right).

Each of the true lines (red, green, blue, black) may come from distinct *units*, i.e. each could represent a different part of the day or a different US state, etc. When there are different effects for each unit, we say that there is *unit heterogeneity* in the data.

In the example above, each line has a different intercept. But the slopes could be different, or both the intercepts and slopes could be different:



In this case, we use a random-effects model because some of the coefficients are random.

For instance, in the first example above, the intercepts varied, in which case the intercept coefficient would be replaced with a random variable α^i drawn from the normal distribution:

$$y = \alpha^i + \beta^i x + \epsilon$$

Or in the case of the slopes varying, we'd say that β^i is a random variable drawn from the normal distribution. In each case, α is the mean intercept and β is the mean slope.

When both slope and intercept vary, we draw them together from a multivariate normal distribution since they may have some relation, i.e.

$$\begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix} \sim \Phi\left(\begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \Sigma\right)$$

Now consider when there are multiple levels of these effects that we want to model. For instance, perhaps there are differences across US states but also differences across US regions.

In this case, we will have a hierarchy of effects. Let's say only the intercept is affected - if we wanted to model the effects of US regions and US states on separate levels, then the α_i will be drawn from a distribution according to the US region, $\alpha_i \sim \Phi(\mu_{\text{region}}, \sigma_{\alpha}^2)$, and then the regional mean which parameterizes α_i 's distribution is drawn from a distribution of regional means, $\mu_{\text{region}} \sim \Phi(\mu, \sigma_r^2)$.

References

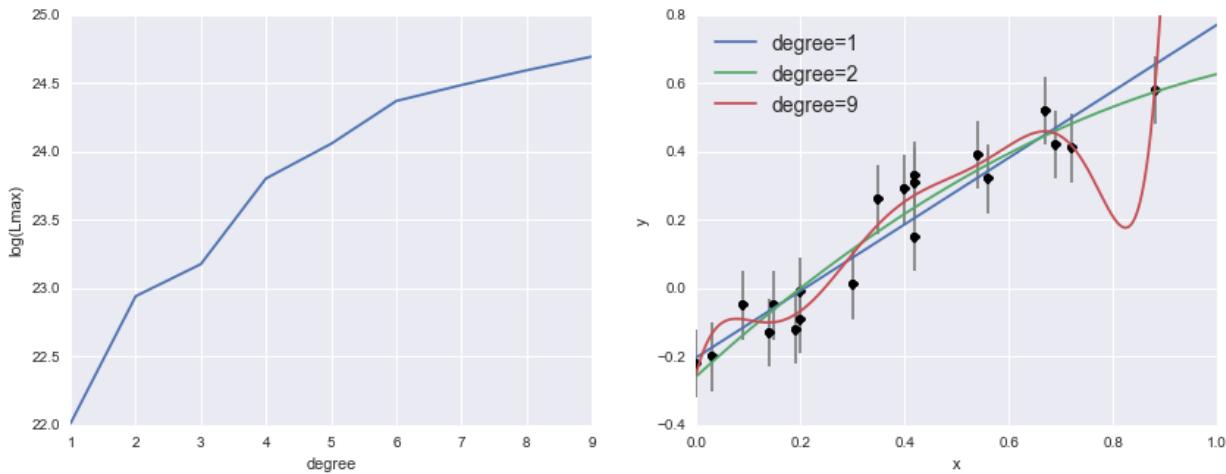
- POLS 509: Hierarchical Linear Models. Justin Esarey. https://www.youtube.com/watch?v=g_4z6o7XZbQ

9.53 Model fitting vs model selection

Model fitting is just about fitting a particular model to data, e.g. minimizing error against it. The resulting fit model might not actually be appropriate for the data - it may overfit it, for instance, or be overly complex. For example, we may fit a high-degree polynomial and a straight line. We might find that the straight line is a better model for the data. The process of choosing a model in this way is *model selection*.

So we need some way of quantifying the quality of models in order to compare them.

A naive approach is to use the likelihood (the product of the probabilities of each datapoint), or more commonly, the log-likelihood (the sum of the log probabilities of each datapoint) and then select the model with the greatest likelihood (this is the maximum likelihood approach). This method is problematic, however, because more complicated (higher-degree) polynomial models will always have a higher likelihood, though they are not necessarily better in the sense that we mean (they overfit the data).



More complex model, greater data likelihood [source](#)

9.53.1 Model fitting

Say you have datapoints x_1, \dots, x_n and errors for those datapoints e_1, \dots, e_n . Say there is some true value for x , we'll call it x_{true} , that we want to learn.

A frequentist approach assumes this true value is fixed and that the data is random. So in this case, we consider the distribution $P(x_i, e_i | x_{\text{true}})$ and want to identify a point estimate - that is, a single value - for x_{true} . This distribution tells us the probability of a point x_i with its error e_i .

For instance, if we assume that x is normally distributed:

$$P(x_i, e_i | x_{\text{true}}) = \frac{1}{\sqrt{2\pi e_i^2}} \exp\left(\frac{-(x_i - x_{\text{true}})^2}{2e_i^2}\right)$$

Then we can consider the likelihood of the data overall by taking the product of the probabilities of each individual datapoint:

$$\mathcal{L}(X, E) = \prod_{i=1}^n P(x_i, e_i | x_{\text{true}})$$

Though typically we work with the log likelihood to avoid underflow errors:

$$\log \mathcal{L}(X, E) = \frac{1}{2} \sum_{i=1}^n (\log(2\pi e_i^2) + \frac{(x_i - x_{\text{true}})^2}{e_i^2})$$

A common frequentist approach to fitting a model is to use maximum likelihood. That is, find an estimate for x_{true} which maximizes this log likelihood:

$$\underset{x_{\text{true}}}{\operatorname{argmax}} \log \mathcal{L}$$

Equivalently, we could minimize the loss (e.g. the squared error).

For simple cases, we can compute the maximum likelihood estimate analytically, by solving $\frac{d \log \mathcal{L}}{dx_{\text{true}}} = 0$

When all the errors e_i are equal, this ends up reducing to:

$$x_{\text{true}} = \frac{1}{n} \sum_{i=1}^n x_i$$

That is, the mean of the datapoints.

For more complex situations, we instead use numerical optimization (i.e. we approximate the estimate).

The Bayesian approach instead involves looking at $P(x_{\text{true}} | x_i, e_i)$, that is, we look at a probability distribution for the unknown value based on fixed data. We aren't looking for a point estimate (a single value) any more, but rather describe x_{true} as a probability distribution. If we do want a point estimate (often you have to have a concrete value to work with), we can take the expected value from the distribution.

$P(x_{\text{true}} | x_i, e_i)$ is computed:

$$P(x_{\text{true}} | x_i, e_i) = \frac{P(x_i, e_i | x_{\text{true}}) P(x_{\text{true}})}{P(x_i, e_i)}$$

Which is to say, it is the posterior distribution. For simple cases, the posterior can be computed analytically, but more often you will need Markov Chain Monte Carlo to approximate it.

9.53.2 Model Selection

Just as model fitting differs between frequentist and Bayesian approaches, so does model selection.

Frequentists compare *model likelihood*, e.g., for two models M_1, M_2 , they would compare $P(D|M_1), P(D|M_2)$.

Bayesians compare the *model posterior*, e.g. $P(M_1|D), P(M_2|D)$.

The parameters are left out in both cases here since we aren't concerned with how good the fit of the model is, but rather, how appropriate the model itself is as a "type" of model.

We can use Bayes theorem to turn the posterior into something we can compute:

$$P(M | D) = P(D | M) \frac{P(M)}{P(D)}$$

Using conditional probability, we know that $P(D | M)$ can be computed as the integral over the parameter space of the likelihood:

$$P(D | M) = \int_{\Omega} P(D | \theta, M) P(\theta | M) d\theta$$

Computing $P(D)$ - the probability of seeing your data *at all* - is really hard, impossible even. But we can avoid dealing with it by comparing $P(M_1 | D)$ and $P(M_2 | D)$ as an odds ratio:

$$O_{21} \equiv \frac{P(M_2 | D)}{P(M_1 | D)} = \frac{P(D | M_2)}{P(D | M_1)} \frac{P(M_2)}{P(M_1)}$$

We still have to deal with $\frac{P(M_2)}{P(M_1)}$, which is known as the *prior odds ratio* (because $P(M_1), P(M_2)$ are priors). This ratio is assumed to equal 1 if there's no reason to believe or no prior evidence that one model will do better than the other.

The remaining ratio $\frac{P(D | M_2)}{P(D | M_1)}$ is known as the *Bayes factor* and is the most important part here. The integrals needed to compute the Bayes factor can be approximated using MCMC.

9.53.3 References

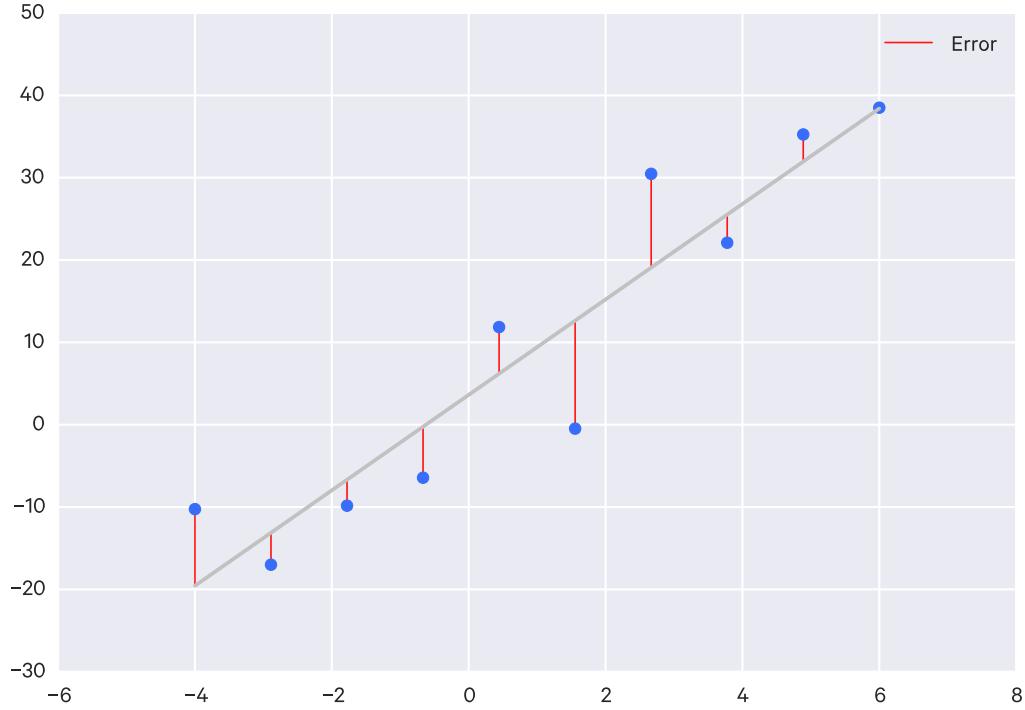
- <https://jakevdp.github.io/blog/2015/08/07/frequentism-and-bayesianism-5-model-selection/>

9.53.4 Squared error of the regression line

For a line $y = mx + b$, the error of a point (x_n, y_n) against that line is:

$$y_n - (mx_n + b)$$

Intuitively, this is the vertical difference between the point on the line at x_n and the actual point at x_n .



Example of error

The **squared error of the line** is the sum of the squares of all of these errors:

$$SE_{\text{line}} = \sum_{i=0}^n (y_i - (mx_i + b))^2$$

To get a best fit line, you want to minimize this squared error. That is, you want to find m and b which minimizes SE_{line} . This works out as ¹:

$$m = \frac{\bar{x}\bar{y} - \bar{x}\bar{y}}{\bar{x}^2 - \bar{x}^2}$$

$$b = \bar{y} - m\bar{x}$$

Note that you can alternatively calculate the regression line slope m as with the covariance and variance:

$$m = \frac{\text{Cov}(x, y)}{\text{Var}(x)}$$

The line that these values yields is the **regression line**.

¹Reminder: a bar over a variable (\bar{x}) means the mean of those values. So $\bar{x}^2 = \frac{x_1^2 + x_2^2 + \dots + x_n^2}{n}$

We can calculate the total variation in y , $SE_{\bar{y}}$, as:

$$SE_{\bar{y}} = \sum_{i=0}^n (y_i - \bar{y})^2$$

And then we can calculate the percentage of total variation in y described by the regression line:

$$1 - \frac{SE_{\text{line}}}{SE_{\bar{y}}}$$

This is known as the **coefficient of determination** or **R-squared**.

The closer R-squared is to 1, the better a fit the line is.

9.54 Residuals

A **residual** e_i is the difference between the observed and predicted outcome, i.e.:

$$e_i = y_i - \hat{y}_i$$

This can also be thought of as the vertical distance between an observed data point and the regression line.

Fitting a line by **least squares** minimizes $\sum_{i=1}^n e_i^2$; that is, it minimizes the **mean squared error** (MSE) between the line and the data. But there always remain some error from the fit line; this remaining error is the residual.

e_i can be interpreted as estimates of the regression error ϵ_i , since we can only compute the true error if we know the true model parameters.

We can measure the quality of a linear model, which is called **goodness of fit**. One approach is to look at the variation of the residuals. You can also use the coefficient of determination (R^2), explained previously, which measures the variance explained by the least squares line.

9.54.1 Residual (error) variation

Residual variation measures how well a regression line fits the data points.

The average squared residual (the estimated residual variance) is the same as the mean squared error, i.e. $\sigma^2 = \frac{1}{n} \sum_{i=1}^n e_i^2$.

However, to make this estimator unbiased, you're more likely to see:

$$\hat{\sigma}^2 = \frac{1}{n-2} \sum_{i=1}^n e_i^2$$

That is, with the degrees of freedom taken into account (here for intercept and slope, which both have to be estimated).

The square root of this estimated variance, σ , is the root mean squared error (RMSE).

9.54.2 Total variation

The total variation is equal to the residual variation (variation after removing the predictor) plus the systematic/regression variation (the variation explained by the regression model):

$$\sum_{i=1}^n (Y_i - \bar{Y})^2 = \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 + \sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2$$

R^2 ($0 \leq R^2 \leq 1$) is the percent of total variability that is explained by the regression model, that is:

$$R^2 = \frac{\text{regression variation}}{\text{total variation}} = \frac{\sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2} = 1 - \frac{\text{residual variation}}{\text{total variation}} = 1 - \frac{\sum_{i=1}^n (Y_i - \hat{Y}_i)^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2}$$

R^2 can be a misleading summary of model fit since deleting data or adding terms will inflate it.

9.54.3 Cross-validation

Cross-validation is just the process of splitting your data such that one portion is used to fit the model and the other portion - the *held-out* subset - is used to test it.

This testing just involves measuring the error of the model on the test data. You can use a variety of error measures; a common one is the *root mean square error*:

$$\text{RMSE}(\hat{\theta}) = \sqrt{\frac{1}{n} \sum_i (\theta - \hat{\theta})^2}$$

The error can also be written more generally as $E[\hat{\theta} - \theta]^2$, and is composed of *variance* and *bias*, which is to say:

$$E[\hat{\theta} - \theta]^2 = \text{Var}(\hat{\theta}) + \text{Bias}(\hat{\theta})^2$$

If the model is underfit, the bias is low and the variance is high. If the model is overfit, the reverse is true. This is a better measure than just looking at the training error, because that error always

improves as more variables (polynomial terms) are added, which can lead to overfitting and overly complex models.

Regular cross-validation still has some issues. Your random subset may not be representative, and the fitted model will have higher variance than a model fit to all the samples.

A enhancement of cross-validation is **k-fold cross-validation**. Here you split the dataset into k smaller sets ("folds").

Then you iterate over each fold, holding it out, training the model on the other folds, then evaluating on the held-out fold. Then you average the error across these iterations.

10

NLP

Much of this is adapted from my notes for the Natural Language Processing course taught by Dan Jurafsky and Christopher Manning at Stanford.

10.1 Challenges

Ambiguity is one of the greatest challenges to NLP:

For example:

Fed raises interest rates, where “raises” is the verb, and “Fed” is the noun phrase Fed
 raises interest rates, where “interest” is the verb, and “Fed raises” is the noun phrase

Other challenges include:

- *non-standard english*: for instance, text shorthand, phrases such as “SOOO PROUD” as opposed to “so proud”, or hashtags, etc
- *segmentation issues*: [the] [New] [York-New] [Haven] [Railroad] vs. [the] [New York]-[New Haven] [Railroad]
- *idioms* (e.g. “get cold feet”, doesn’t literally mean what it says)
- *neologisms* (e.g. “unfriend”, “retweet”, “bromance”)
- *world knowledge* (e.g. “Mary and Sue are sisters” vs “Mary and Sue are mothers.”)
- *tricky entity names*: “Where is A Bug’s Life playing”, or “a mutation on the *for* gene”

The typical approach is to codify knowledge about language & knowledge about the world and find some way to combine them to build probabilistic models.

10.2 Terminology

Synset

A synset is a set of synonyms that represent a single sense of a word.

Wordform

The full inflected surface form: e.g. “cat” and “cats” are different wordforms.

Lemma

The same stem, part of speech, rough word sense: e.g. “cat” and “cats” are the same lemma.

One lemma can have many meanings.

For example:

a *bank* can hold investments... agriculture on the east *bank*...

These usages have a different *sense*.

Sense

A discrete representation of an aspect of a word's meaning.

Homonyms

Words that share form but have unrelated, distinct meanings (such as “bank”).

- *Homographs*: bank/bank, bat/bat
- *Homophones*: write/right, piece/peace

Polysemy

the *bank* was built in 1875 (“bank” = a building belonging to a financial institution) I withdrew money from the *bank* (“bank” = a financial institution)

A *polysemous* word has *related* meanings.

Systematic polysemy, or *metonymy*, is when the meanings have a *systematic* relationship.

For example, “school”, “university”, “hospital” - all can mean the institution or the building, so the systematic relationship here is building <=> organization.

Another example is author <=> works of author, e.g. “Jane Austen wrote Emma” and “I love Jane Austen”.

Synonyms

Different words that have the same *propositional* meaning in some or all contexts. However, there may be no examples of *perfect synonymy* since even if propositional meaning is identical, they may vary in notions of politeness or other usages and so on.

For example, “water” and “H₂O” - each are more appropriate in different contexts.

As another example, “big” and “large” - sometimes they can be swapped, sometimes they cannot:

That's a big plane. How large is that plane? (Acceptable) Miss Nelson became kind of a big sister to Benjamin. Miss Nelson became kind of a large sister to Benjamin (Not as acceptable)

The latter works less because “big” has multiple senses, one of which does not correspond to “large”.

Antonyms

Senses which are opposite with respect to one feature of meaning, but otherwise are similar, such as dark/light, short/fast, etc.

Hyponym

One sense is a hyponym of another if the first sense is more specific (i.e. denotes a subclass of the other).

- *car* is a hyponym of *vehicle*
- *mango* is a hyponym of *fruit*

Hypernym/Superordinate

- *vehical* is a hypernym of *car*
- *fruit* is a hypernym of *mango*

Token

An instance of that type in running text; N = number of tokens, i.e. counting every word in the sentence, regardless of uniqueness.

Type

An element of the vocabulary; V = vocabulary = set of types ($|V|$ = the size of the vocabulary), i.e. counting every unique word in the sentence.

10.3 Data preparation

10.3.1 Sentence segmentation

“!”, “?” are pretty reliable indicators that we’ve reached the end of a sentence. Periods can mean the end of the sentence *or* an abbreviation (e.g. Inc. or Dr.) or numbers (e.g. 4.3).

10.3.2 Tokenization

The best approach for tokenization varies widely depending on the particular problem and language. German, for example, has many long compound words which you may want to split up. Chinese has no spaces (no easy way for *word segmentation*), Japanese has no spaces and multiple alphabets.

10.3.3 Normalization

Once you have your tokens you need to determine how to normalize them. For example, “USA” and “U.S.A.” could be collapsed into a single token. But about “Windows”, “window”, and “windows”?

Some common approaches include:

- *case folding* - reducing all letters to lower case (but sometimes case may be informative)
- *lemmatization* - reduce inflections or variant forms to base form.
- *stemming* = reducing terms of their stems; a crude chopping of affixes; a simplified version of lemmatization. The Porter stemmer is the most common English stemmer.

10.3.4 Term Frequency-Inverse Document Frequency (tf-idf) Weighting

Using straight word counts may not be the best approach in many cases.

Rare terms are typically more informative than frequent terms, so we want to bias our numerical representations of tokens to give rarer words higher weights. We do this via *inverse document frequency weighting* (idf):

$$idf_t = \log\left(\frac{N}{df_t}\right)$$

For a term t which appears in df documents (df_t = document frequency for t).

\log is used here to “dampen” the effect of idf.

This can be combined with t ’s term frequency tf_d for a particular document d to produce tf-idf weighting, which is the best known weighting scheme for text information retrieval:

$$w_{t,d} = (1 + \log tf_{t,d}) \times \log\left(\frac{n}{df_t}\right)$$

10.3.5 The Vector Space Model (VSM)

This representation of text data - that is, some kind of numerical feature for each word, such as the tf-idf weight and frequency, defines a $|V|$ -dimensional vector space (where V is the vocabulary size).

- *terms* are the axes of space
- *documents* are points (vectors) in this space
- this space is *very high-dimensional* when dealing with large vocabularies
- these vectors are very *sparse* - most entries are zero

10.3.6 Normalizing vectors

This is a different kind of normalization than the previously mentioned one, which was about normalizing the language. Here, we are normalizing vectors in a more mathematical sense.

Vectors can be length-normalized by dividing each of its components by its length. We can use the L2 norm, which makes it a *unit vector* ("unit" means it is of length 1):

$$\|\vec{x}\|_2 = \sqrt{\sum_i x_i^2}$$

This means that if we have, for example, a document and copy of that document with every word doubled, length normalization causes each to have identical vectors (without normalization, the copy would have been twice as long).

10.4 Measuring similarity between text

10.4.1 Minimum edit distance

The *minimum edit distance* between two strings is the minimum number of editing operations (insertion/deletion/substitution) needed to transform one into the other. Each editing operation has a cost of 1, although in *Levenshtein minimum edit distance* substitutions cost 2 because they are composed of a deletion and an insertion.

10.4.2 Jaccard coefficient

The Jaccard coefficient is a commonly-used measure of overlap for two sets A and B .

$$\text{jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

A set has a Jaccard coefficient of 1 against itself: $\text{jaccard}(A, A) = 1$.

If A and B have no overlapping elements, $jaccard(A, B) = 0$.

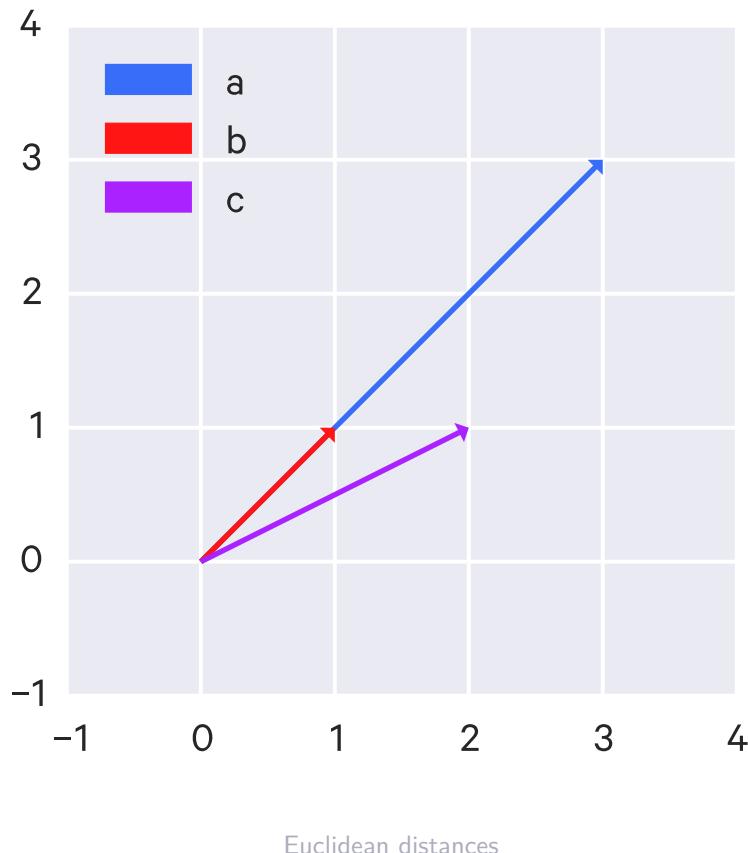
The Jaccard coefficient does *not* consider term frequency, just set membership.

10.4.3 Euclidean Distance

Using the vector space model above, the similarity between two documents can be measured by the euclidean distance between their two vectors.

However, euclidean distance can be problematic since longer vectors have greater distance.

For instance, there could be one document vector, a , and another document vector b which is just a scalar multiple of the first document. Intuitively they may be more similar since they lie along the same line. But by euclidean distance, c is closer to a .



10.4.4 Cosine similarity

In cases like the euclidean distance example above, using *angles* between vectors can be a better metric for similarity.

For length-normalized vectors, cosine similarity is just their dot product:

$$\cos(\vec{q}, \vec{d}) = \vec{q} \cdot \vec{d} = \sum_{i=1}^{|V|} q_i d_i$$

Where q and d are length-normalized vectors and q_i is the tf-idf weight of term i in document q and d_i is the tf-idf weight of term i in document d .

10.5 Probabilistic Language Models

The approach of probabilistic language models involves generating some probabilistic understanding of language - what is likely or unlikely.

These probabilistic models have applications in many areas:

- Machine translation: $P(\text{high winds tonight}) > P(\text{large winds tonight})$.
- Spelling correction: $P(\text{about fifteen minutes from}) > P(\text{about fifteen minuets from})$.
- Speech recognition: $P(\text{I saw a van}) > P(\text{eyes awe of an})$.

So generally you are asking: what is the probability of this given sequence of words?

You could use the *chain rule* here:

$$\begin{aligned} P(\text{the water is so transparent}) = \\ P(\text{the}) \times P(\text{water}|\text{the}) \times P(\text{is}|\text{the water}) \\ \times P(\text{so}|\text{the water is}) \times P(\text{transparent}|\text{the water is so}) \end{aligned}$$

Formally, the above would be expressed:

$$P(w_1 w_2 \dots w_n) = \prod_i P(w_i | w_1 w_2 \dots w_{i-1})$$

Note that probabilities are usually done in *log space* to avoid *underflow*, which occurs if you're multiplying many small probabilities together, and because then you can just add the probabilities, which is faster than multiplying:

$$p_1 \times p_2 \times p_3 = \log p_1 + \log p_2 + \log p_3$$

To make estimating these probabilities manageable, we use the *Markov assumption* and assume that a given word's conditional probability only depends on the immediately preceding k words, *not* the entire preceding sequence:

$$P(w_1 w_2 \dots w_n) \approx \prod_i P(w_i | w_{i-k} \dots w_{i-1})$$

10.5.1 n-grams

The *unigram* model treats each word as if they have an independent probability:

$$P(w_1 w_2 \dots w_n) \approx \prod_i P(w_i)$$

The *bigram* model conditions on the previous word:

$$P(w_1 w_2 \dots w_{i-1}) \approx \prod_i P(w_i | w_{i-1})$$

We estimate bigram probabilities using the *maximum likelihood estimate* (MLE):

$$P_{MLE}(w_i | w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$$

Which is just the count of word i occurring after word $i - 1$ over all of the occurrences of word $i - 1$.

This can be extended to trigrams, 4-grams, 5-grams, etc.

Though language has *long-distance dependencies*, i.e. the probability of a word can depend on another word much earlier in the sentence, n-grams work well in practice.

Dealing with zeros

Zeroes occur if some n-gram occurs in the testing data which didn't occur in the training set.

Say we had the following training set:

... denied the reports ... denied the claims ... denied the request

And the following test set:

... denied the offer

Here $P(\text{offer} | \text{denied the}) = 0$ since the model has not encountered that term.

We can get around this using *Laplace smoothing*, also known as *_add-one smoothing*): simply pretend that we saw each word once more than we actually did (i.e. add one to all counts).

With add-one smoothing, our MLE becomes:

$$P_{Add-1}(w_i | w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i) + 1}{\text{count}(w_{i-1}) + V}$$

Note that this smoothing can be very blunt and may drastically change your counts.

Interpolation

With interpolation, you mix unigrams, bigrams, and trigrams, assigning weights to each.

Simple linear interpolation looks like:

$$P(w_n | w_{n-1} w_{n-2}) = \lambda_1 P(w_n | w_{n-1} w_{n-2}) + \lambda_2 P(w_n | w_{n-1}) + \lambda_3 P(w_n)$$

Such that $\sum_i \lambda_i = 1$. The λ parameters here are the different weights for the different n-grams.

If you want to get more complex, you can have the λ parameters vary by context:

$$P(w_n | w_{n-1} w_{n-2}) = \lambda_1(w_{n-2}^{n-1}) P(w_n | w_{n-1} w_{n-2}) + \lambda_2(w_{n-2}^{n-1}) P(w_n | w_{n-1}) + \lambda_3(w_{n-2}^{n-1}) P(w_n)$$

The process of determining the λ parameters usually involves holding out part of the training corpus, then getting the n-gram probabilities from the remaining training data, then selecting the λ parameters to maximize the probability of the held-out data:

$$\log P(w_1 \dots w_n | M(\lambda_1 \dots \lambda_k)) = \sum \log P_{M(\lambda_1 \dots \lambda_k)}(w_i | w_{i-1})$$

10.6 Text Classification

The general text classification problem is given an input document d and a fixed set of classes $C = \{c_1, c_2, \dots, c_j\}$ output a predicted class $c \in C$.

10.6.1 Naive Bayes

This supervised approach to classification is based on Bayes' rule. It relies on a very simple representation of the document called “bag of words”, which is ignorant of the sequence or order of word occurrence (and other things), and only pays attention to their counts/frequency.

So you can represent the problem with Bayes' rule:

$$P(c|d) = \frac{P(d|c)P(c)}{P(d)}$$

And the particular problem at hand is finding the class which maximizes $P(c|d)$, that is:

$$C_{MAP} = \operatorname{argmax}_{c \in C} P(c|d) = \operatorname{argmax}_{c \in C} P(d|c)P(c)$$

Where C_{MAP} is the maximum a posteriori class.

Using our bag of words assumption, we represent a document as features x_1, \dots, x_n without concern for their order:

$$C_{MAP} = \operatorname{argmax}_{c \in C} P(x_1, x_2, \dots, x_n | c)P(c)$$

We additionally assume *conditional independence*, i.e. that the presence of one word doesn't have any impact on the probability of any other word's occurrence:

$$P(x_1, x_2, \dots, x_n | c) = P(x_1 | c) \cdot P(x_2 | c) \cdot \dots \cdot P(x_n | c)$$

And thus we have the *multinomial naive bayes classifier*:

$$C_{NB} = \operatorname{argmax}_{c \in C} P(c_j) \prod_{x \in X} P(x | c)$$

To calculate the prior probabilities, we use the *maximum likelihood estimates* approach:

$$P(c_j) = \frac{\operatorname{doccount}(C = c_j)}{N_{doc}}$$

That is, the prior probability for a given class is the count of documents in that class over the total number of documents.

Then, for words:

$$P(w_i | c_j) = \frac{\operatorname{count}(w_i, c_j)}{\sum_{w \in V} \operatorname{count}(w, c_j)}$$

That is, the count of a word in documents of a given class, over the total count of words in that class.

To get around the problem of zero probabilities (for words encountered in test input but not in training, which would cause a probability of a class to be zero since the probability of a class is the joint probability of the words encountered), you can use Laplace smoothing (see above):

$$P(w_i | c_j) = \frac{\operatorname{count}(w_i, c_j) + 1}{(\sum_{w \in V} \operatorname{count}(w, c_j)) + |V|}$$

Note that to avoid underflow (from multiplying lots of small probabilities), you may want to work with log probabilities (see above).

In practice, even with all these assumptions, Naive Bayes can be quite good:

- Very fast, low storage requirements
- Robust to irrelevant features (they tend to cancel each other out)
- Very good in domains with many equally important features
- Optimal if independence assumptions hold
- A good, dependable baseline for text classification

10.6.2 Evaluating text classification

The possible outcomes are:

- true positive: correctly identifying something as true
- false positive: incorrectly identifying something as true
- true negative: correctly identifying something as false
- false negative: incorrectly identifying something as false

The *accuracy* of classification is calculated as:

$$\text{accuracy} = \frac{tp + tn}{tp + fp + fn + fn}$$

Though as a metric it isn't very useful if you are dealing with situations where the correct class is sparse and most words you encounter are not in the correct class:

Say you're looking for a word that only occurs 0.01% of the time. you have a classifier you run on 100,000 docs and the word appears in 10 docs (so 10 docs are correct, 99,990 are not correct). but you can have that classifier classify all docs as not correct and get an amazing accuracy of $99,990/100,000 = 99.99\%$ but the classifier didn't actually do anything!

So other metrics are needed.

Precision measures the percent of selected items that are correct:

$$\text{precision} = \frac{tp}{tp + fp}$$

Recall measures the percent of correct items that are selected:

$$\text{recall} = \frac{tp}{tp + fn}$$

Typically, there is a trade off between recall and precision - the improvement of one comes at the sacrifice of the other.

The *F measure* combines both precision and recall into a single metric:

$$F = \frac{1}{\alpha \frac{1}{P} + (1 - \alpha) \frac{1}{R}} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}$$

Where α is a weighting value so you can assign more importance to either precision or recall.

People usually use the *balanced F1 measure*, where $\beta = 1$ (that is, $\alpha = 1/2$):

$$F = \frac{2PR}{P + R}$$

10.7 Named Entity Recognition (NER)

Named entity recognition is the extraction of *entities* - people, places, organizations, etc - from a text.

Many systems use a combination of statistical techniques, linguistic parsing, and gazetteers to maximize detection recall & precision. Distant supervision and unsupervised techniques can also help with training, limiting the amount of gold-standard data necessary to build a statistical model.

Boundary errors are common in NER:

First *Bank of Chicago* announced earnings...

Here, the extractor extracted “Bank of Chicago” when the correct entity is the “First Bank of Chicago”.

A general NER approach is to use supervised learning:

1. Collect a set of training documents
2. Label each entity with its entity type or 0 for “other”.
3. Design feature extractors
4. Train a sequence classifier to predict the labels from the data.

10.8 Relation Extraction

International Business Machines Corporation (IBM or the company) was incorporate in the State of New York on June 16, 1911, as the Computing-Tabulating-Recording Co. (C-T-R)...

From such a text you could extract the following *relation triples*:

Founder-year(IBM, 1911)
 Founding-location(IBM, New York)

These relations may be represented as *resource description framework (RDF) triples* in the form of subject predicate object.

Golden Gate Park location San Francisco

10.8.1 Ontological Relations

- IS-A describes a subsumption between classes, called a *hypernym*:

Giraffe IS-A ruminant IS-A ungulate IS-A mammal IS-A vertebrate IS-A animal...

- instance-of relation between individual and class

San Francisco instance-of city

There may be many domain-specific ontological relations as well, such as founded (between a PERSON and an ORGANIZATION), cures (between a DRUG and a DISEASE), etc.

10.8.2 Methods

Relation extractors can be built using:

- handwritten patterns
- supervised machine learning
- semi-supervised and unsupervised
 - bootstrapping (using seeds)
 - distance supervision
 - unsupervised learning from the web

Handwritten patterns

- Advantages:
 - can take advantage of domain expertise
 - human patterns tend to be high-precision
- Disadvantages:
 - human patterns are often low-recall
 - hard to capture all possible patterns

Supervised

- Advantages:
 - can get high accuracy if...
 - * there's enough hand-labeled training data
 - * if the test is similar enough to training
- Disadvantages:
 - labeling a large training set is expensive
 - don't generalize well

You could use classifiers: find all pairs of named entities, then use a classifier to determine if the two are related or not.

Unsupervised

If you have no training set and either only a few seed tuples or a few high-precision patterns, you can *bootstrap* and use the seeds to accumulate more data.

The general approach is:

1. Gather a set of seed pairs that have a relation R
2. Iterate:
 1. Find sentences with these pairs
 2. Look at the context between or around the pair
 3. Generalize the context to create patterns
 4. Use these patterns to find more pairs

For example, say we have the seed tuple $\langle \text{Mark Twain}, \text{Elmira} \rangle$. We could use Google or some other set of documents to search based on this tuple. We might find:

- “Mark Twain is buried in Elmira, NY”
- “The grave of Mark Twain is in Elmira”
- “Elmira is Mark Twain’s final resting place”

which gives us the patterns:

- “X is buried in Y”
- “The grave of X is in Y”
- “Y is X’s final resting place”

Then we can use these patterns to search and find more tuples, then use those tuples to find more patterns, etc.

Two algorithms for this bootstrapping is the Dipre algorithm and the Snowball algorithm, which is a version of Dipre which requires the strings be named entities rather than any string.

Another semi-supervised algorithm is *distance supervision*, which mixes bootstrapping and supervised learning. Instead of a few seeds, you use a large database to extract a large number of seed examples and go from there:

1. For each relation R
2. For each tuple in a big database
3. Find sentences in a large corpus with both entities of the tuple
4. Extract frequent contextual features/patterns
5. Train a supervised classifier using the extracted patterns

10.9 Sentiment Analysis

In general, sentiment analysis involves trying to figure out if a sentence/doc/etc is positive/favorable or negative/unfavorable; i.e. detecting *attitudes* in a text.

The attitude may be

- a simple weighted polarity (positive, negative, neutral), which is more common
- from a set of types (like, love, hate, value, desire, etc)

When using multinomial Naive Bayes for sentiment analysis, it's often better to use *binarized* multinomial Naive Bayes under the assumption that word occurrence matters more than word frequency: seeing "fantastic" five times may not tell us much more than seeing it once. So in this version, you would cap word frequencies at one.

An alternate approach is to use $\log(freq(w))$ instead of 1 for the count.

However, sometimes raw word counts don't work well either. In the case of IMDB ratings, the word "bad" appears in more 10-star reviews than it does in 2-star reviews!

Instead, you'd calculate the *likelihood* of that word occurring in an n -star review:

$$P(w|c) = \frac{f(w, c)}{\sum_{w \in C} f(w, c)}$$

And then you'd used the *scaled likelihood* to make these likelihoods comparable between words:

$$\frac{P(w|c)}{P(w)}$$

10.9.1 Sentiment Lexicons

Certain words have specific sentiment; there are a variety of sentiment lexicons which specify those relationships.

10.9.2 Challenges

Negation

“I *didn’t* like this movie” vs “I *really* like this movie.”

One way to handle negation is to prefix every word following a negation word with NOT_, e.g. “I didn’t NOT_like NOT_this NOT_movie”.

“Thwarted Expectations” problem

For example, a film review which talks about how great a film *should* be, but fails to live up to those expectations:

This film should be *brilliant*. It sounds like a *great* plot, the actors are *first grade*, and the supporting cast is *good* as well, and Stallone is attempting to deliver a good performance. However, it *can’t hold up*.

10.10 Summarization

Generally, summarization is about producing an abridged version of a text without or with minimal loss of important information.

There are a few ways to categorize summarization problems.

- Single-document vs multi-document summarization: summarizing a single document, yielding an abstract or outline or headline, or producing a gist of the content of multiple documents?
- Generic vs query-focused summarization: give a general summary of the document, or a summary tailored to a particular user query?
- Extractive vs abstractive: create a summary from sentences pulled from the document, or generate new text for the summary?

Here, extractive summarization will be the focus (abstractive summarization is really hard).

The *baseline* used in summarization, which often works surprisingly well, is just to take the first sentence of a document.

10.10.1 The general approach

Summarization usually uses this process:

1. Content Selection: choose what sentences to use from the document.

- You may weight salient words based on tf-idf, its presence in the query (if there is one), or based on topic signature.
 - For the latter, you can use *log-likelihood ratio* (LLR):

$$\text{weight}(w_i) = \begin{cases} 1 & \text{if } -2 \log \lambda(w_i) > 10 \\ 0 & \text{otherwise} \end{cases}$$

- Weight a sentence (or a part of a sentence, i.e. a *window*) by the weights of its words:

$$\text{weight}(s) = \frac{1}{|S|} \sum_{w \in S} \text{weight}(w)$$

- You can combine LLR with *maximal marginal relevance* (MMR), which is a greedy algorithm which selects sentences by their similarity to the query and by their dissimilarity (novelty) to already-selected sentences to avoid redundancy.

2. Information Ordering: choose the order for the sentences in the summary.

- If you are summarizing documents with some chronological order to them, such as the news, then it makes sense to order sentences chronologically (if you are, for example, summarizing a set of news articles).
- You can also use *topical ordering*, and order sentences by the order of topics in the source documents.
- You can also use *coherence*:
 - Choose orderings that make neighboring sentences (cosine) similar.
 - Choose orderings in which neighboring sentences discuss the same entity.

3. Sentence Realization: clean up the sentences so that the summary is coherent or remove unnecessary content. You could remove:

- *appositives*: “Rajam[, an artist living in Philadelphia], found inspiration in the back of city magazines.”
- *attribution clauses*: “Sources said Wednesday”
- *initial adverbials*: “For example”, “At this point”

11

Neural Nets

When it comes down to it, a neural net is just a very sophisticated way of fitting a curve, so you shouldn't be dazzled just by the mention of it - it's not a magic solution. A lot of a neural net's effectiveness depends on, for instance, how you represent your input parameters.

11.1 Biological basis

Artificial neural networks (ANNs) are based off of biological neural networks such as the human brain. Neural networks are composed of *neurons* which send signals to each other in response to certain inputs.

A single neuron takes in one or more inputs (via dendrites), processes it, and fires one output (via its axon).

Note that the term “unit” is often used instead of “neuron” when discussing artificial neural networks to dissociate these from the biological version - while there is some basis in biological neural networks, there are vast differences, so it is a deceit to present them as analogous.

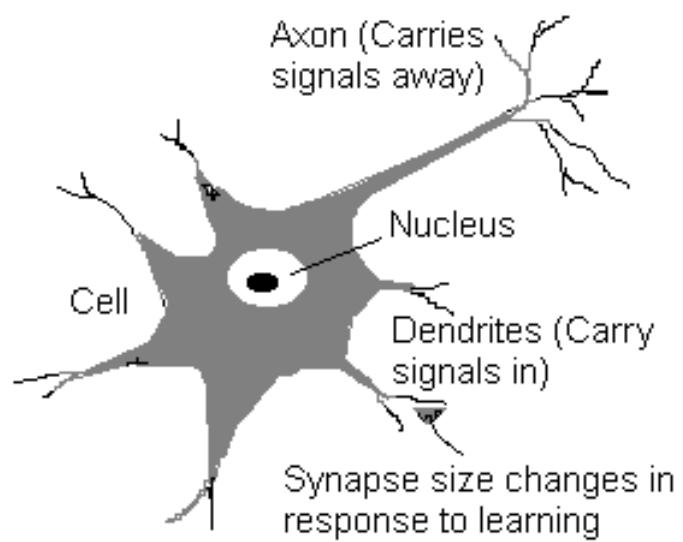
11.2 Perceptron: a simple artificial neuron

A *perceptron*, first described by Frank Rosenblatt in 1957, is an artificial neuron (a computational model of a biological neuron, first introduced in 1943 by Warren McCulloch and Walter Pitts). It too has multiple inputs, processes them, and returns one output.

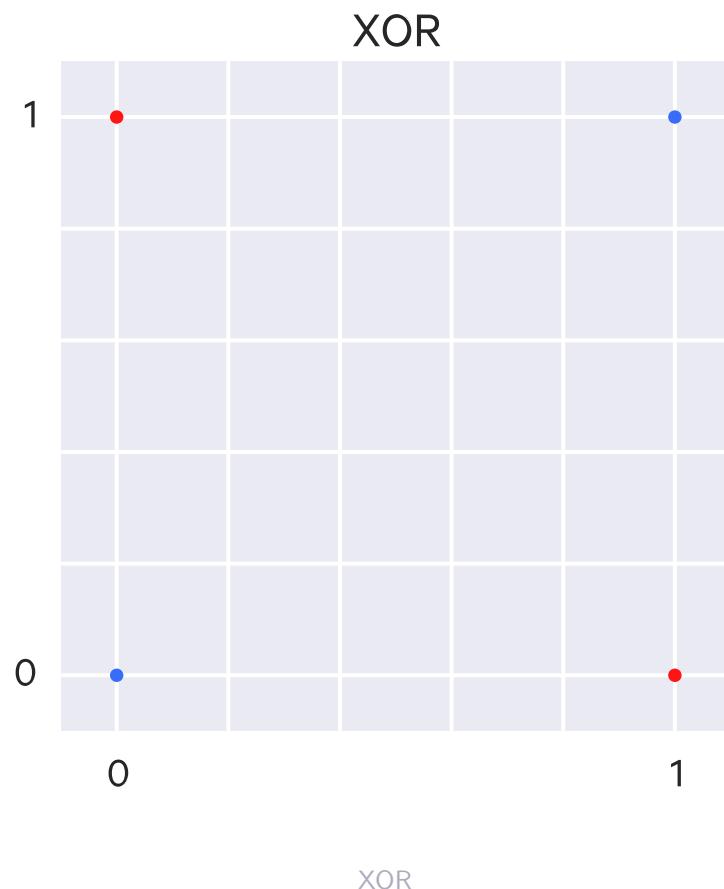
Each input has a weight associated with it.

In the simplest artificial neuron, a “binary” or “classic spin” neuron, the neuron “fires” an output of “1” if the weighted sum of these inputs is above some *threshold*, or “-1” if otherwise.

A single-layer perceptron can't learn XOR:



Source: <http://ulcar.uml.edu/~iag/CS/Intro-to-ANN.html>



A line can't be drawn to separate the *As* from the *Bs*; that is, this is not a linearly separable problem. Single-layer perceptrons cannot represent linearly inseparable functions.

11.2.1 Activation functions

The function that determines the output is known as the *activation function*. In the binary/classic spin case, it might look like:

```
weights = [...]
inputs  = [...]
sum_w = sum([weights[i] * inputs[i] for i in range(len(inputs))])

def activate(sum_w, threshold):
    return 1 if sum_w > threshold else -1
```

Or:

$$\text{output} = \begin{cases} -1 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Note that $w \cdot x = \sum_j w_j x_j$, so it can be notated as a dot product where the weights and inputs are vectors.

In some interpretations, the “binary” neuron returns “0” or “1” instead of “-1” or “1”.

An activation function can generally be described as some function:

$$\text{output} = f(w \cdot x + b)$$

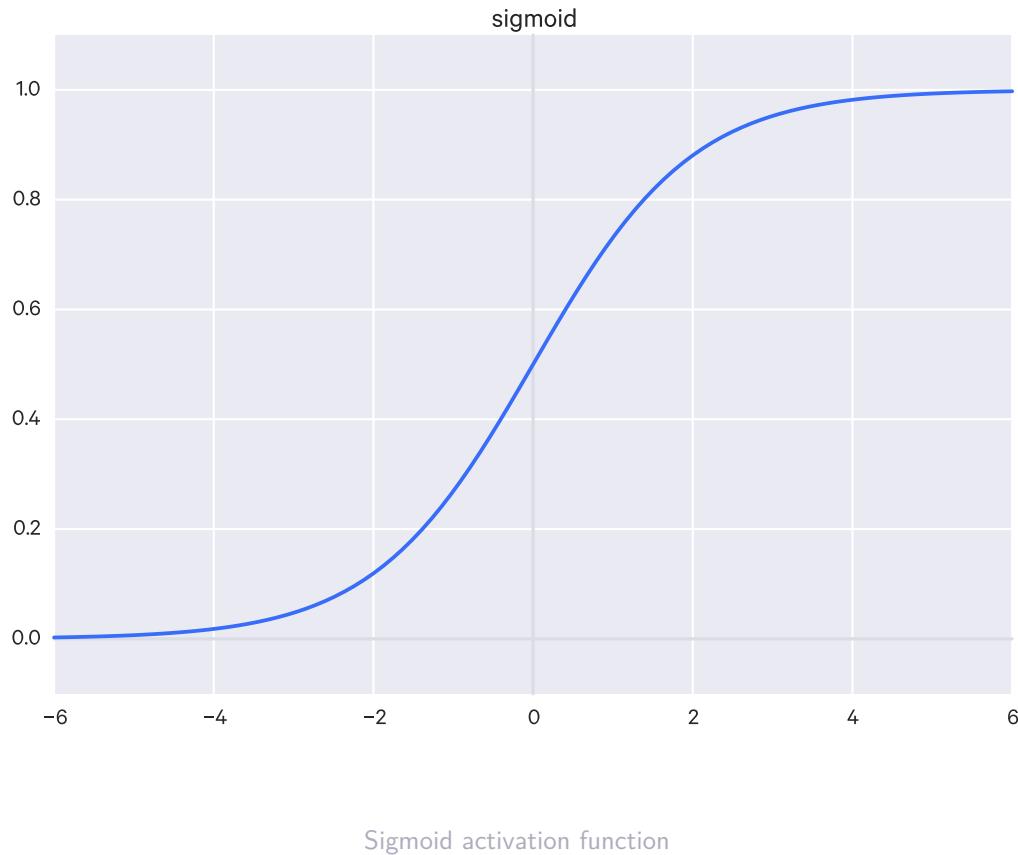
where b is the bias (see below).

Common activation functions

A common activation function is the *sigmoid* function, which takes input and squashes it to be in $[0, 1]$, it has the form:

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

However, the sigmoid activation function has some problems. If the activation yields values at the tails of 0 or 1, the gradient ends up being almost 0. In backpropagation, this local gradient is multiplied with the gradient of the node's output against the total error - if this local gradient is near 0, it “kills” the gradient preventing any signal from going further backwards in the network. For



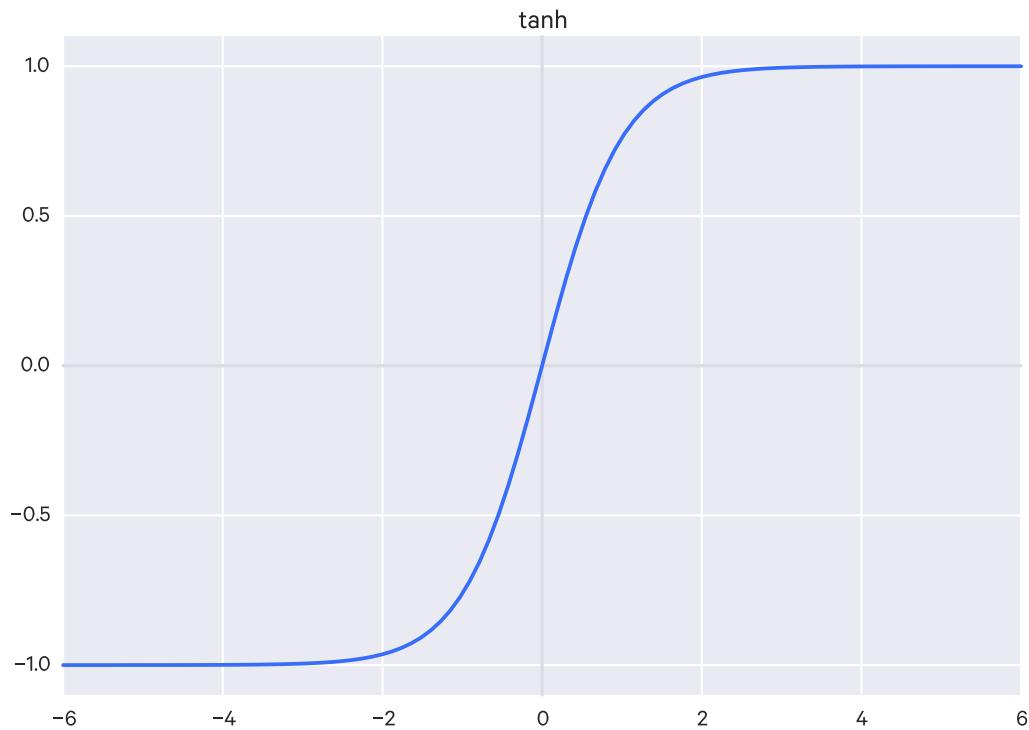
this reason, when using the sigmoid activation function you must be careful of how you initialize the weights - if they are too large, you will “saturate” the network and kill the gradient in this way.

Furthermore, sigmoid outputs are not zero-centered:

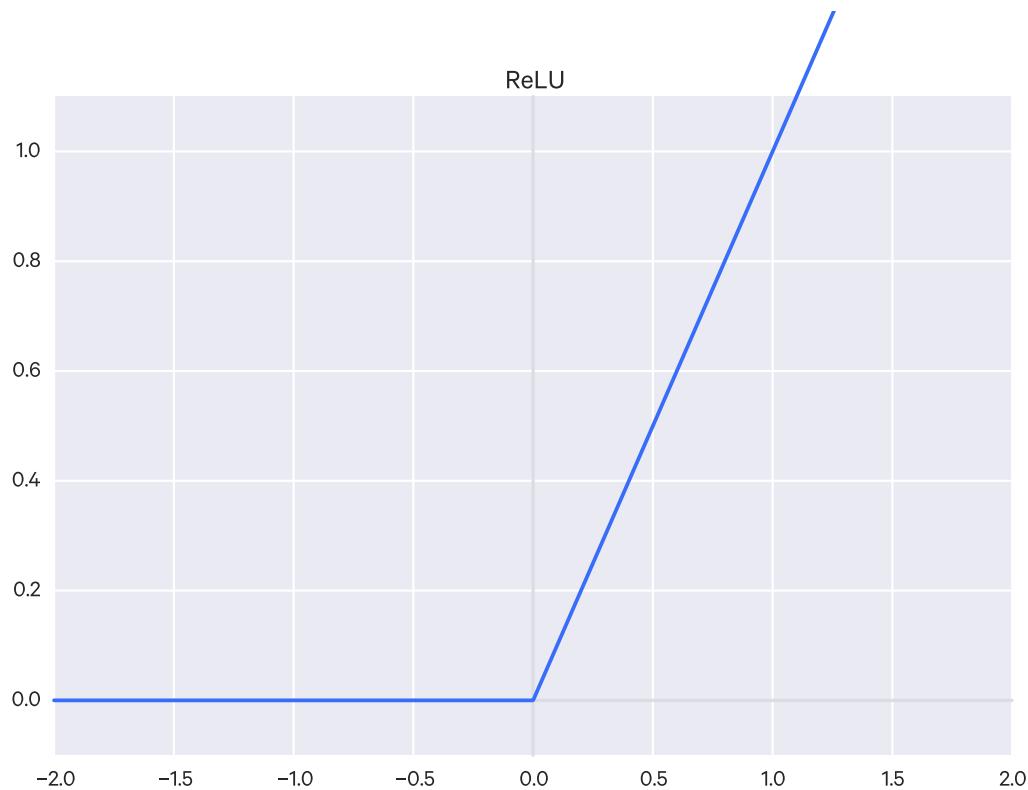
This is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ elementwise in $f = wTx + b$), then the gradient on the weights w will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above. [source](#) (TODO revisit this/clarify)

The \tanh activation function is another option; it squishes values to be in $[-1, 1]$. However, while its output is zero-centered, it suffers from the same activation saturation issue that the sigmoid does.

The Rectified Linear Unit (ReLU) is $f(x) = \max(0, x)$, that is, it just thresholds at 0. Compared to the sigmoid/tanh functions, it converges with stochastic gradient descent quickly. Though there is not the same saturation issue as with the sigmoid/tanh functions, ReLUs can still “die” in a different sense - their weights can be updated such that the neuron never activates again, which causes the



tanh activation function



ReLU activation function

gradient through that neuron to be zero from then on, thus resulting in the same “killing” of the gradient as with sigmoid/tanh. In practice, lowering the learning rate can avoid this.

Leaky ReLUs are an attempt to fix this problem. Rather than outputting 0 when $x < 0$, there will instead be a small negative slope (~ 0.01) when $x < 0$. However, it does not always work well.

There are also some units which defy the conventional activation form of output = $f(w \cdot x + b)$. One is the *Maxout* neuron. Its function is $\max(w_1^T x + b_1, w_2^T x + b_2)$, which is a generalization of the ReLU and the leaky ReLU (both are special forms of Maxout). It has the benefits of ReLU but does not suffer the dying ReLU problem, but its main drawback is that it doubles the number of parameters for each neuron (since there are two weight vectors and two bias units).

Karpathy suggests:

Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of “dead” units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

[source](#)

References

- CS231n Convolutional Neural Networks for Visual Recognition, Module 1: Neural Networks Part 1: Setting up the Architecture. Andrej Karpathy. <https://cs231n.github.io/neural-networks-1/>

11.2.2 Bias

The activation threshold is often expressed instead as a perceptron’s *bias* b , where bias == -threshold:

$$\text{output} = \begin{cases} -1 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

A lower bias means that a stronger signal is necessary for the perceptron to fire.

11.3 Multilayered Perceptron (MLP, Feed-Forward ANN)

A *multilayered perceptron* (MLP) is a simple neural network with an input layer, and output layer, and one or more intermediate layers of neurons.

When we describe the network in terms of layers as a “ N -layer” neural network, we leave out the input layer (i.e. a 1-layer neural network has an input and an output layer, a 2-layer one has an input, a hidden, and an output layer.). ANNs may also be described by its number of nodes (units), or, more commonly, by the number of parameters in the entire network. (CS231n Convolutional Neural

Networks for Visual Recognition, Module 1: Neural Networks Part 1: Setting up the Architecture.
Andrej Karpathy. <https://cs231n.github.io/neural-networks-1/>)

Two layer networks are limited to convex functions (TODO how are layers counted? are input and outputs counted in Neural Computing: Theory and Practice?), whereas 3+ layer networks have no such limitation.

This model is often called *feed-forward* because values go into the input layer and are fed into subsequent layers.

Different *learning algorithms* (such as backpropagation, detailed below) can train such a network so that its weights are adjusted appropriately for a given task. It's worth emphasizing that the *structure* of the network is distinct from the *learning algorithm* which tunes its weights and biases.

11.3.1 Training a perceptron via “backpropagation”

The most common algorithm for adjusting a perceptron's weights and biases is the *back-propagation of error*:

- The initial NN state does not matter; the weights and biases may be random
- Training data is input into the NN to the output neurons, in feed-forward style
- The error of the output is then propagated backwards (from the output layer back to the input layer).
- As the error is propagated, weights and biases are adjusted to minimize the remaining error between the actual and desired outputs
- The amount weights and biases are adjusted is determined by a *delta rule* or *delta function*. This often involves some constant which manages the “momentum” of learning. This learning constant can help “jiggle” the network out of local optima, but you want to take care that it isn't set so high that the network will also jiggle out of the global optima. As a simple example:

```
# LEARNING_CONSTANT is defined elsewhere
def adjust_weight(weight, error, input):
    return weight + error * input * LEARNING_CONSTANT
```

- In some cases, a *simulated annealing* approach is used, where the learning constant may be tempered (made smaller, less drastic) as the network evolves, to avoid jittering the network out of the global optima.

Notes on backpropagation:

- a backpropagation NN (BPNN), like most machine learning models, is capable of overfitting (becoming tuned to a very specific training data; lacking any ability to generalize to new input data).
- BPNNs are very much a “black box” in that you don't really know what's going on in the intermediary layers
- training can be slow, but it usually isn't too bad on fast machines

11.4 Backpropagation

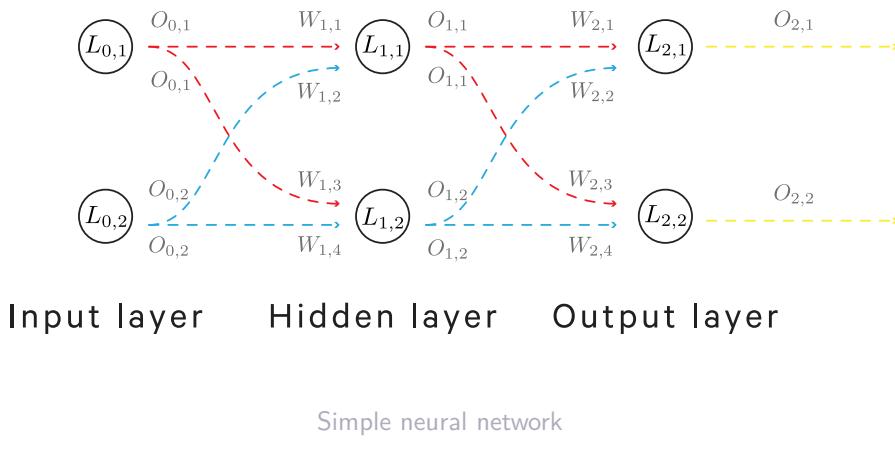
Backpropagation is just the calculation of partial derivatives (the gradient) by moving backwards through the network (from output to input), accumulating them by applying the chain rule. “Backpropagation” is almost just a special term for the chain rule in the context of training neural networks. This is because a neural network can be thought of as a composition of functions, in which case to compute the derivative of the overall function, you just apply the chain rule for computing derivatives.

Side note: composition of functions as in, each layer represents a function taking in the inputs of the previous layer’s output, e.g. if the previous layer is a function that outputs a vector, $g(x)$, then the next layer, if we call it a function f , is $f(g(x))$.

Backpropagation is key because it is how deep neural networks (multilayer perceptrons) learn. Backpropagation computes the gradient of the loss function with respect to the weights in the network (i.e. the derivatives of the loss function with respect to each weight in the network) in order to update the weights.

We compute the total error for the network on the training data and then want to know how a change in an individual weight within the network affects this total error (i.e. the result of our cost function), e.g. $\frac{\partial E_{\text{total}}}{\partial w_i}$.

Consider the following simple neural net:

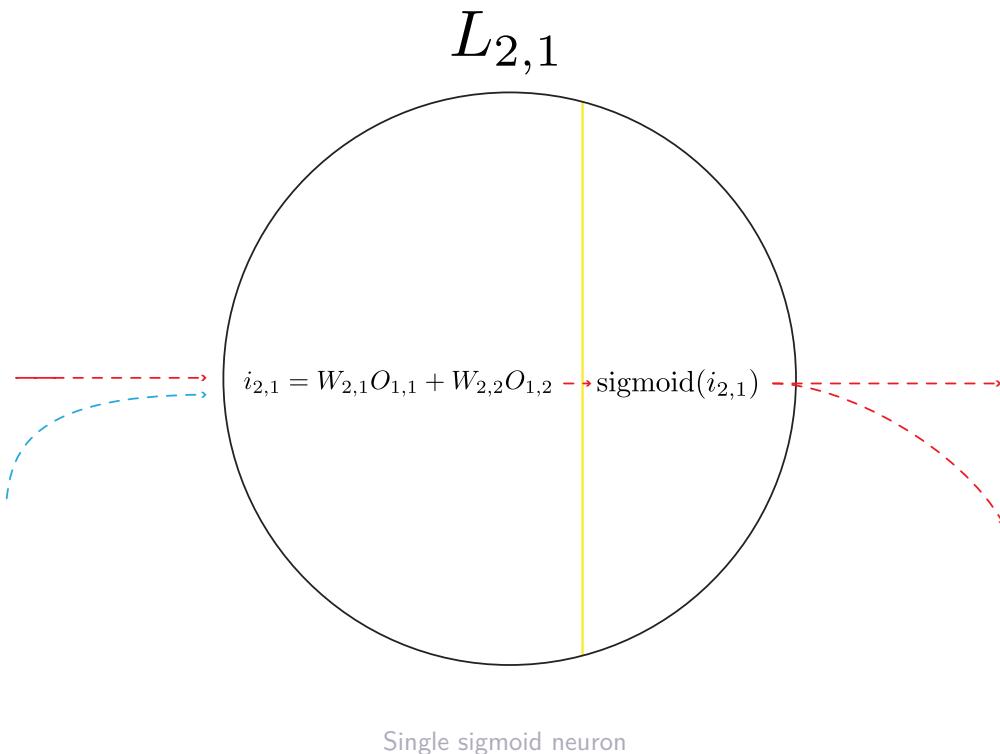


Here’s a single neuron expanded:

Remember that a neuron processes its inputs by computing the dot product of its weights and inputs (i.e. the sum of its weight-input products) and then passes this resulting *net input* into its activation function (in this case, it is the sigmoid function).

Say we have passed some training data through the network and computed the total error as E_{total} . To update the weight $w_{2,1}$, for example, we are looking for the partial derivative $\frac{\partial E_{\text{total}}}{\partial w_{2,1}}$, which by the chain rule is equal to:

$$\frac{\partial E_{\text{total}}}{\partial w_{2,1}} = \frac{\partial E_{\text{total}}}{\partial o_{2,1}} \times \frac{\partial o_{2,1}}{\partial i_{2,1}} \times \frac{\partial i_{2,1}}{\partial w_{2,1}}$$



Then we take this value and subtract it, multiplied by a learning rate η (sometimes notated α), from the current weight $w_{2,1}$ to get $w_{2,1}$'s updated weight, though updates are only actually applied after these update values have been computed for all of the network's weights.

If we wanted to calculate the update value for $w_{1,1}$, we do something similar:

$$\frac{\partial E_{\text{total}}}{\partial w_{1,1}} = \frac{\partial E_{\text{total}}}{\partial o_{1,1}} \times \frac{\partial o_{1,1}}{\partial i_{1,1}} \times \frac{\partial i_{1,1}}{\partial w_{1,1}}$$

Any activation function can be used with backprop, it just must be differentiable anywhere.

11.4.1 References

- A Step by Step Backpropagation Example. Matt Mazur. March 17, 2015. <http://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>
- Gradient Descent with Backpropagation. July 31, 2015. <http://outlace.com/Beginner-Tutorial-Backpropagation/>

11.4.2 Alternate explanation of the chain rule

(adapted from the CS231n notes cited below)

Refresher on derivatives: say you have a function $f(x, y, z)$. The derivative of f with respect to x is called a *partial derivative*, since it is only with respect to one of the variables, is notated $\frac{\partial f}{\partial x}$ and is just a function that tells you how much f changes due to x at any point. The gradient is just a vector of these partial derivatives, so that there is a partial derivative for each variable (i.e. here it would be a vector of the partial derivative of f wrt x , and then wrt y , and then wrt z).

As a simple example, consider the function $f(x, y) = xy$. The derivatives here are just $\frac{\partial f}{\partial x} = y$, $\frac{\partial f}{\partial y} = x$. What does this mean? Well, take $\frac{\partial f}{\partial x} = y$. This means that, at any given point, increasing x by a infinitesimal amount will change the output of the function by y times the amount that x changed. So if $y = -3$, then any small change in x will decrease f by that amount times -3 .

Now consider the function $f(x, y, z) = (x+y)z$. We can derive this by declaring $q = x+y$ and then re-writing f to be $f = qz$. We can compute the gradient of f in this form (note that it is the same as $f(x, y) = xy$ from before): $\frac{\partial f}{\partial q} = z$, $\frac{\partial f}{\partial z} = q$. The gradient of q is also simple: $\frac{\partial q}{\partial x} = 1$, $\frac{\partial q}{\partial y} = 1$. We can combine these gradients to get the gradient of f wrt to x, y, z instead of wrt to q, z as we have now. We can get the missing partial derivatives wrt to x and y by using the chain rule, which just requires that we multiply the appropriate partials:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}, \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

In code (adapted from the CS231 notes cited below)

```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
dqdx = 1.
dqdy = 1.

# now backprop through q = x + y
dfdxdx = dqdx * dfdq # dq/dx = 1. And the multiplication here is the chain rule!
dfdxdy = dqdy * dfdq # dq/dy = 1
```

So essentially you can decompose any function into smaller, simpler functions, compute the gradients for those, then use the chain rule to aggregate them into the original function's gradient.

References

- CS231n Convolutional Neural Networks for Visual Recognition, Module 1: Backpropagation, Intuitions. Andrej Karpathy. <https://cs231n.github.io/optimization-2/>

11.5 Choosing the network configuration

The general structure of a neural network is input layer \rightarrow 0 or more hidden layers \rightarrow output layer.

Neural networks always have one input layer, and the size of that input layer is equal to the input dimensions (i.e. one node per feature), though sometimes you may have an additional bias node.

Neural networks always have one output layer, and the size of that output layer depends on what you're doing. For instance, if your neural network will be a regressor (i.e. for a regression problem), then you'd have a single output node (unless you're doing multivariate regression). Same for binary classification. However with softmax (more than just two classes) you have one output node per class label, with each node outputting the probability the input is of the class associated with the node.

How do you know how many hidden layers to have? How do you know what size each hidden layer should be?

If your data is linearly separable, then you don't need any hidden layers (and you probably don't need a neural network either and a linear or generalized linear model may be plenty).

Neural networks with additional hidden layers become difficult to train; networks with multiple hidden layers are the subject of deep learning. For many problems, one hidden layer suffices, and you may not see any performance improvement from adding additional hidden layers.

A rule of thumb for deciding the size of the hidden layer is that the size should be between the size between the input size and output size (for example, the mean of their sizes).

11.6 Recurrent neural networks (Feed-Back ANN)

A *recurrent neural network* is a *feed-back* neural network, that is, it is an ANN where the outputs of neurons are fed back into their inputs, continuing until stopped externally (or just continuing for a specific duration). They are less common than feedforward ANNs but have properties which may give them advantages over feedforward ANNs for certain problems.

Feedforward networks are contrasted to recurrent networks (recurrent networks are just feedforward networks with some feedback loops).

11.7 Sigmoid neurons (aka logistic neurons)

A sigmoid neuron is another artificial neuron, similar to a perceptron. However, while the perceptron has a binary output, the sigmoid neuron has a continuous output, $\sigma(w \cdot x + b)$, defined by a special activation function known as the *sigmoid function* σ (also known as the *logistic function*):

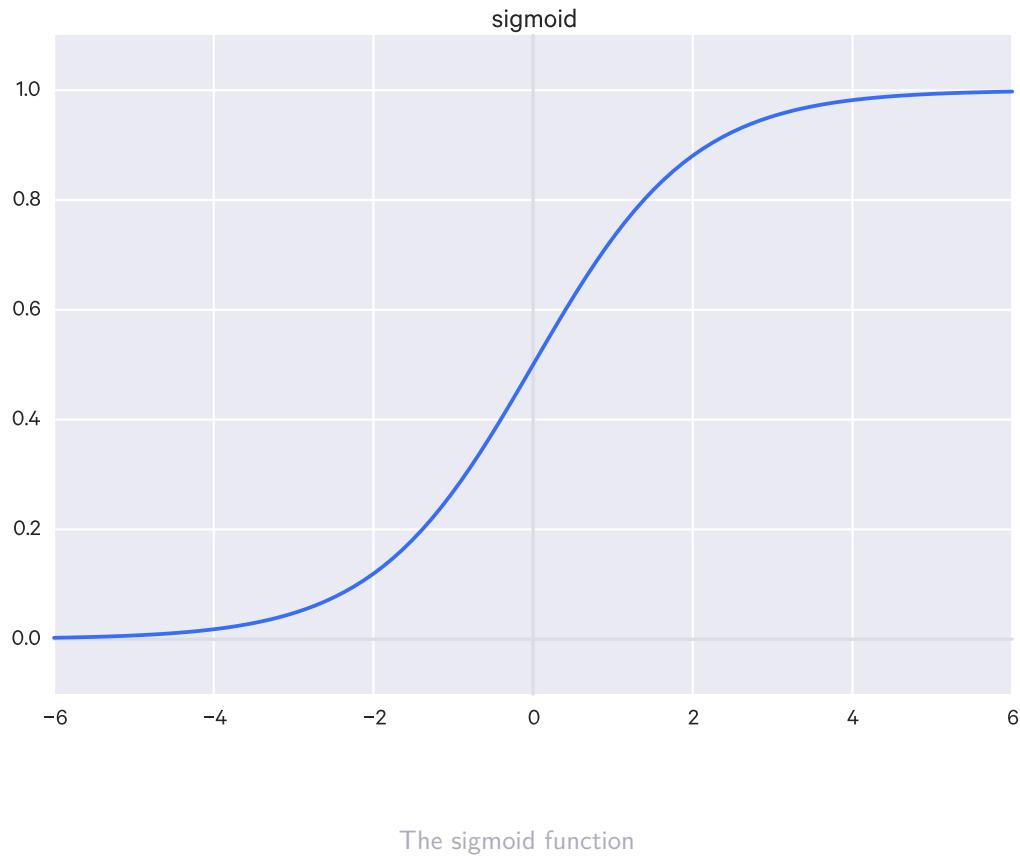
$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}.$$

which can also be written:

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}.$$

Note that if $z = w \cdot x + b$ is a large positive number, then $e^{-z} \approx 0$ and thus $\sigma(z) \approx 1$. If z is a large negative number, then $e^{-z} \rightarrow \infty$ and thus $\sigma(z) \approx 0$. So at these extremes, the sigmoid neuron behaves like a perceptron.

Here is the sigmoid function visualized:



Which is a smoothed out step function (which is how a perceptron operates):

Sigmoid neurons are useful because small changes in weights and biases will only produce small changes in output from a given neuron (rather than switching between binary output values, which may be too drastic).

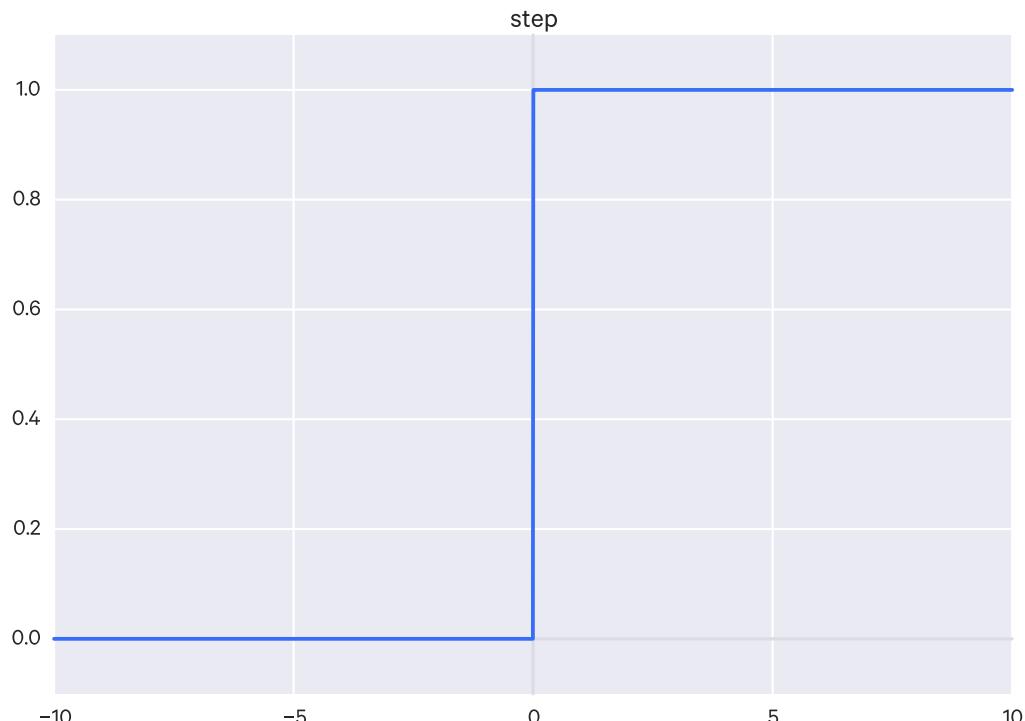
11.8 Cost/loss/objective/error functions

To determine the error, a cost (or “loss” or “objective” or “error”) function is used. A common one is the *quadratic* cost function, also known as *mean squared error* (MSE):

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

where

- w = all weights in the network



The step function

- b = all biases in the network
- n = the number of training inputs
- x = an input vector
- a = the vector of outputs from the network when x is input
- $y(x)$ = the desired output vector

When $C(w, b) \approx 0$, then $a \approx y(x)$.

You want to optimize (minimize) $C(w, b)$. See [Optimization](#) for some approaches. A very common one is gradient descent.

Note that this also averages the costs by including $\frac{1}{n}$. Sometimes just the sum is used, e.g. in cases where the number of training inputs are not known or if more training data is being provided in real-time.

Note that this cost function takes into account *all* training inputs. If using gradient descent, you may want to opt for the stochastic variant (stochastic gradient descent) in which the cost function considers only a random subset (a “mini-batch”) of the training examples.

11.9 RBF (neural) network

You can base your activation function off of [Radial Basis Functions \(RBFs\)](#):

$$f(X) = \sum_{i=1}^N a_i p(||b_i X - c_i||)$$

where

- X = input vector of attributes
- p = the RBF
- c = vector center (peak) of the RBF
- a = the vector coefficient/weight for each RBF
- b = the vector coefficient/weight for each input attribute

11.9.1 Radial Basis Functions (RBF)

A radial basis function (RBF) is a function which is:

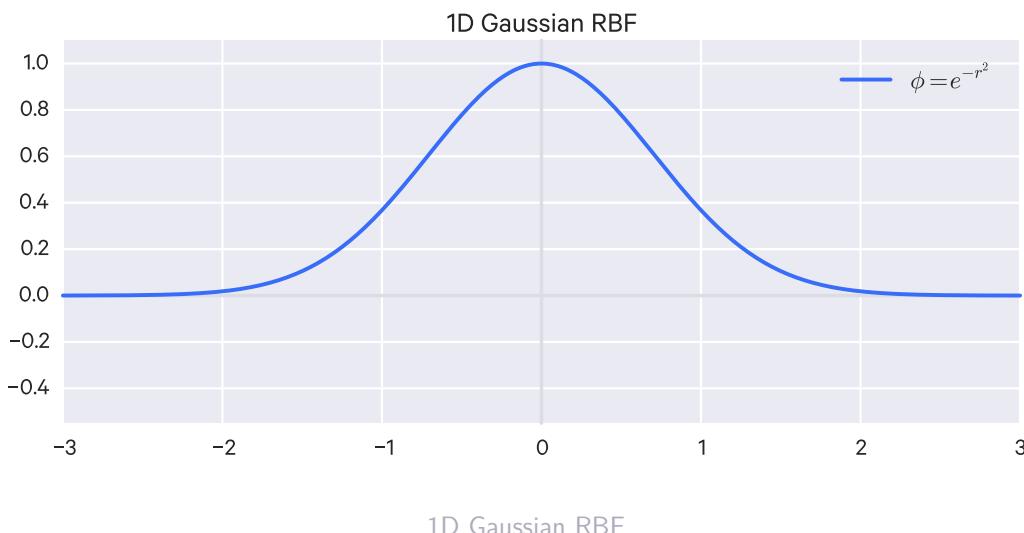
- symmetric about its center, which is its peak (with a value of 1)
- can be in n dimensions, but always returns a single scalar value r , the distance (usually Euclidean) b/w the input vector and the RBF's peak:

$$r = ||x - x_i||$$

ϕ is used to denote a RBF.

Examples

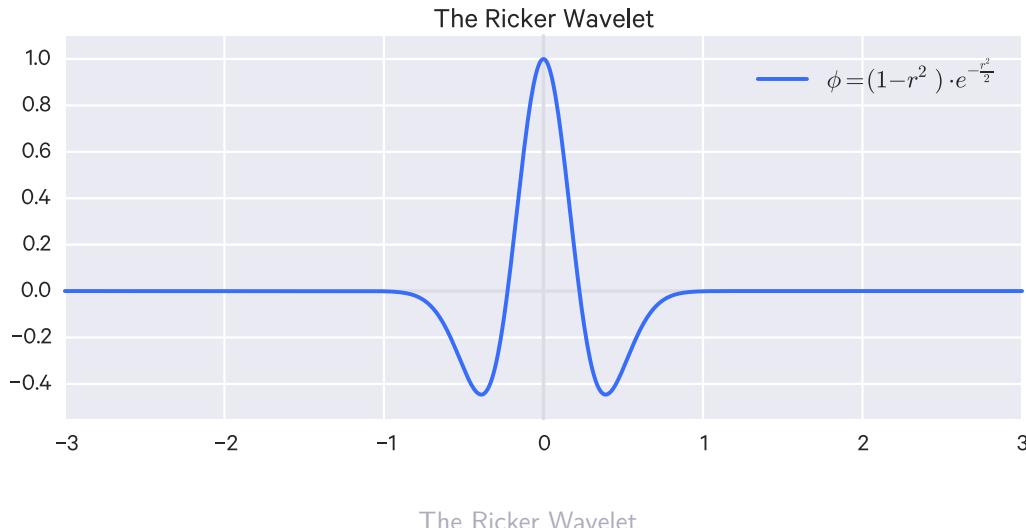
1D Gaussian RBF:



Defined as:

$$\phi(r) = e^{-r^2}$$

The Ricker Wavelet



Defined as:

$$\phi(r) = (1 - r^2) \cdot e^{-\frac{r^2}{2}}$$

11.10 Deep neural networks

Many problems can be broken down into subproblems, each of which can be addressed by a separate neural network.

Say for example we want to know whether or not a face is in an image. We could break that down (*decompose it*) into subproblems like:

- is there an eye?
- is there an ear?
- is there a nose?
- etc.

We could train a neural network on each of these subproblems. We could even break these subproblems further (e.g. “Is there an eyelash?”, “Is there an iris?”, etc) and train neural networks for those, and so on.

Then if we want to identify a face, we can aggregate these networks into a larger network.

This kind of multi-layered neural net is a *deep neural network*.

Multilayer nns must have nonlinear activation functions, otherwise they are equivalent to a single layer network aggregating its weights.

That is, a 2 layer network has weight vectors W_1 and W_2 and input X . The network computes $(XW_1)W_2$, which is equivalent to $X(W_1W_2)$, so the network is equivalent to a single layer network with weight vectors W_1W_2

Deep networks are harder to train - a simple stochastic gradient descent + backpropagation approach is not as effective or quick.

Newer techniques (2006 onward) based on SGD and backpropagation allow deeper (5-10 hidden layers) and larger networks to be effectively trained.

These deep networks can perform much better than shallow networks (networks with just one hidden layer) because they can embody a complex hierarchy of concepts.

11.11 Sources

- *Crash Introduction to Artificial Neural Networks*, Ivan Galkin: <http://ulcar.uml.edu/~iag/CS/Intro-to-ANN.html>
- <http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html>
- *The Nature of Code*, Daniel Shiffman: <http://natureofcode.com/book/chapter-10-neural-networks>
- *Neural Networks and Deep Learning*, Michael Nielsen: <http://neuralnetworksanddeeplearning.com>
- *Neural Networks*, Christos Stergiou & Dimitrios Siganos: http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html
- <http://www.ai-junkie.com/ann/evolved/nnt1.html>

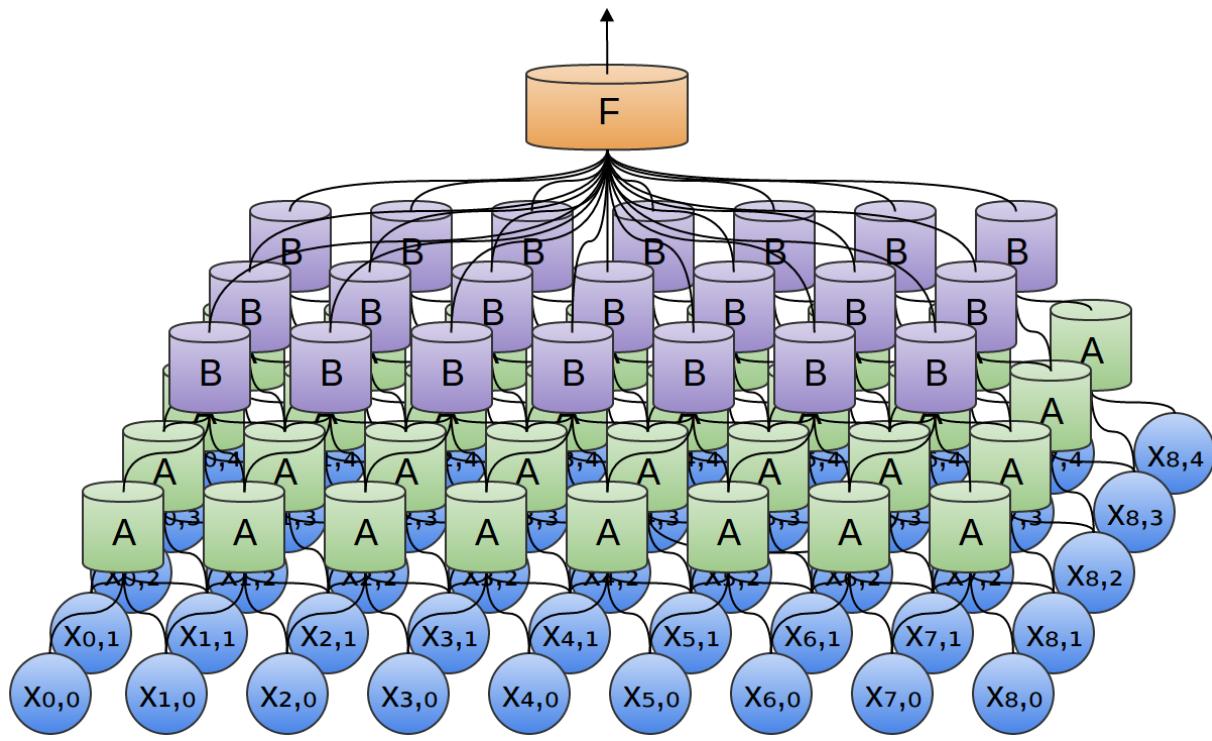
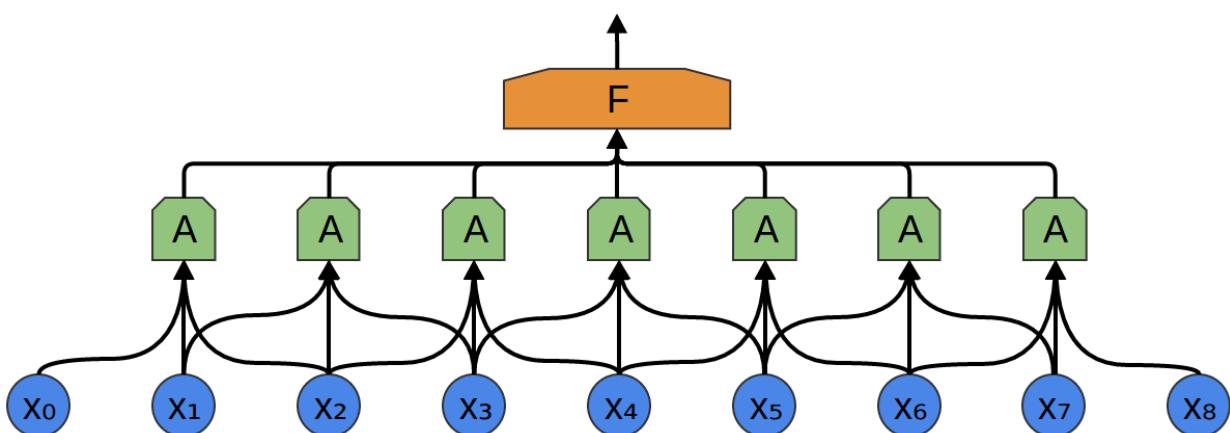
11.12 Convolutional Neural Networks (CNN)

A *convolutional neural network* is one in which multiple copies of the same neuron (i.e. “copies” meaning they share the same weights) as used to form *convolutional layers*, which allows for many neurons which share parameters, thus keeping the number of parameters relatively small.

A convolutional layer computes features which then may be fed into further convolutional layers or to a fully-connected layer.

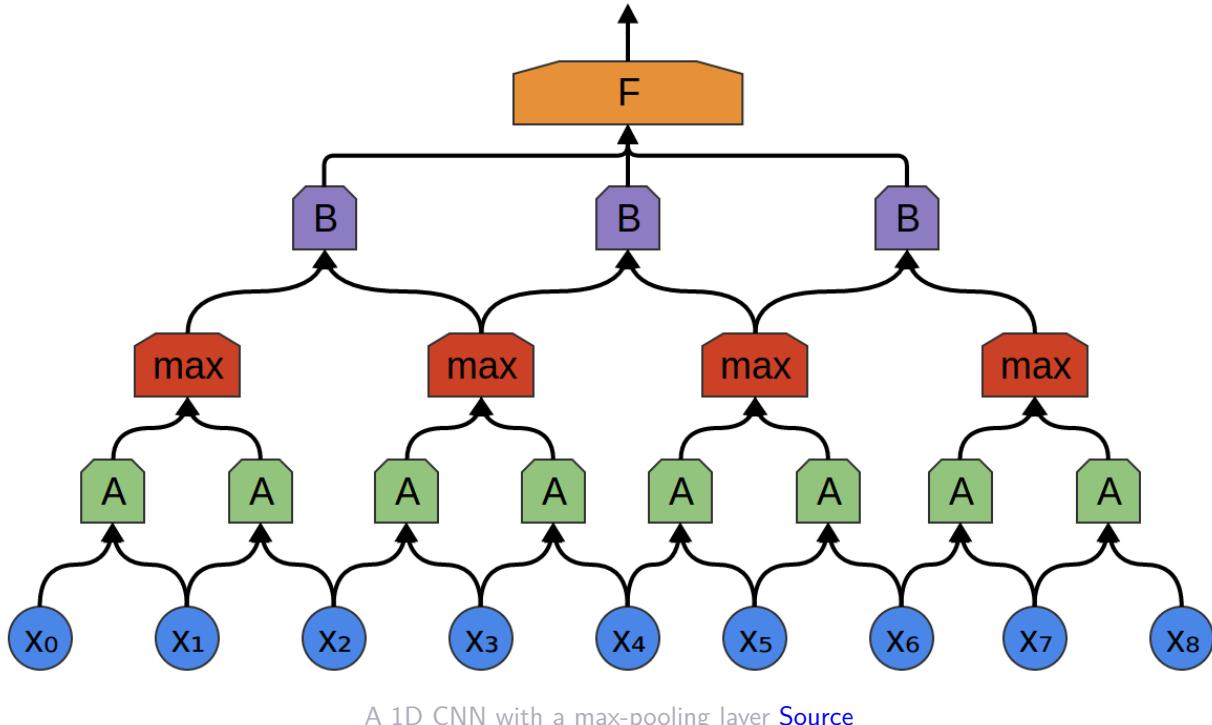
The figure shows a 2D convolutional neural net with inputs layer x and two convolutional layers A and B , consisting of copied neurons (different for each layer), which then feed into a fully-connected layer F .

A convolutional layer operates on a *window* of inputs from its preceding layer; that is, it takes as input multiple neurons from the preceding layer, and these windows overlap. The figure below shows a 1D neural net with a window size of 3 for the A convolutional layer.

A 2D convolutional neural net [Source](#)A is a convolutional layer with a window size of 3 [Source](#)

Often there may be *pooling layers* between convolutional layers. The intuition here is that you may not need a fine-grained resolution of your inputs, so you can group them in some way into a single representative input. For instance, with image recognition - you may not care if something is one or two pixels off, so you might pool those features together.

A popular pooling layer is the *max-pooling layer*, which output the maximum of features in its window. This allows convolutional layers to look at larger sections of data (i.e. patches) and more invariant to small transformations of the data.



In this figure there is a max-pooling layer between the convolutional layers A and B . The maximum of each pair of A fed into the max-pooling layer will be passed along to B .

Typically the units of A are composed of single neurons in parallel, like below.

But each unit in A could also be a mini-network of its own (i.e. have multiple layers of neurons, or a “network-in-network”), as depicted below.

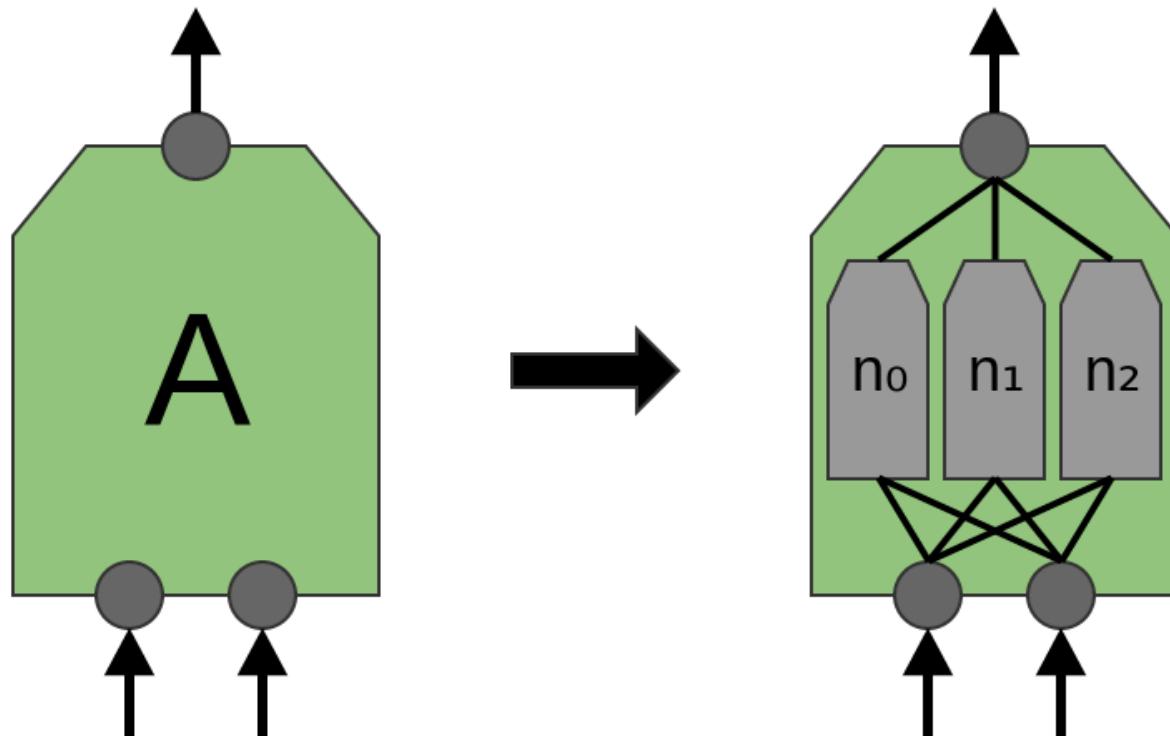
11.12.1 Convolutions

CNNs are based off of *convolutions* from mathematics.

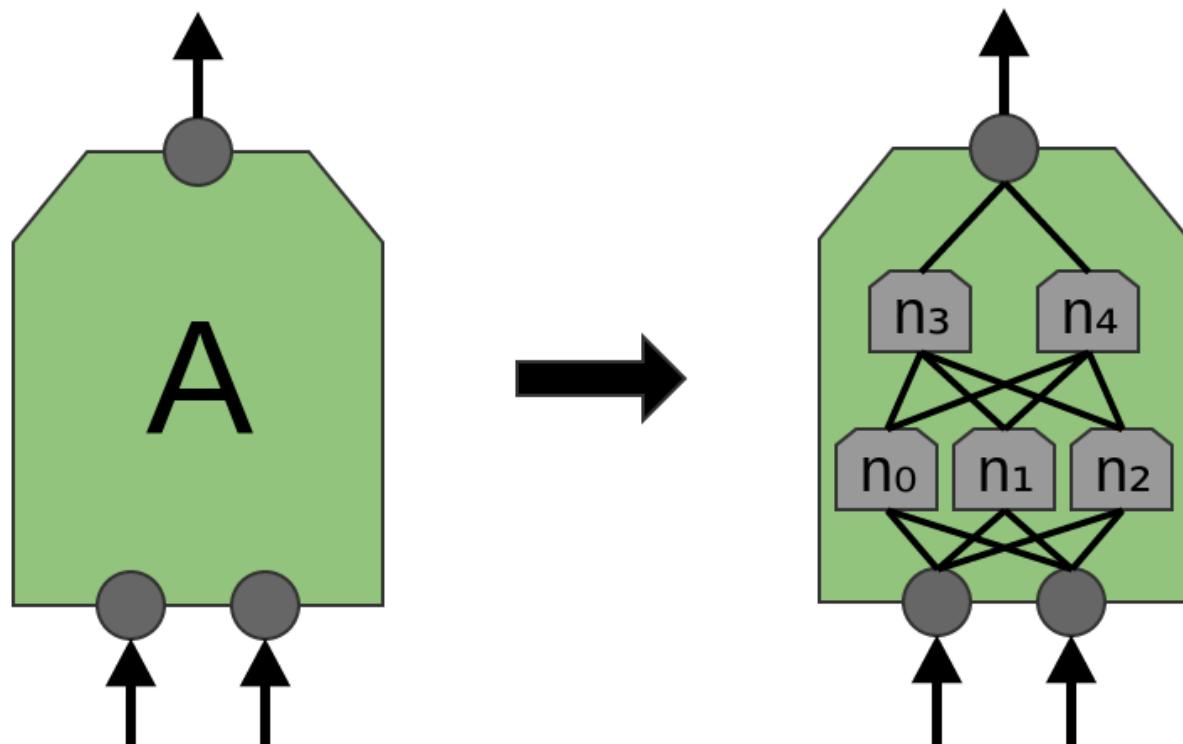
Say you have a ball which you kick once, then kick it again once it stops. On the first kick, the ball moves a distance a with some probability; on the second kick, it moves a distance b with some probability. You want the ball’s total distance to be c .

What’s the probability that you get the distance c ?

Well say $c = 3$. How can $a + b = c$? You could have $a = 1, b = 2$, and the probability of this happening is $f(a) \cdot g(b)$. But this isn’t the only combination of $a + b$ which leads to 3. You could have $a = 0, b = 3$ or $a = 0.5, b = 2.5$ and so on. To find the *total likelihood* of the ball reaching a distance c , you have to sum the probabilities of each of these combinations:



Units of A are usually just neurons in parallel [Source](#)



Units of A could be mini-NNs of their own [Source](#)

$$(f * g)(c) = \sum_{a+b=c} f(a) \cdot g(b)$$

This is a convolution - specifically, it is the convolution of f and g , evaluated at c .

We know that $b = c - a$ so we can re-write this as:

$$(f * g)(c) = \sum_{a+b=c} f(a) \cdot g(c - a)$$

Convolutions can be applied to any number of dimensions, using the same equation as above - it's just that a, b, c become vectors.

For example, in two dimensions:

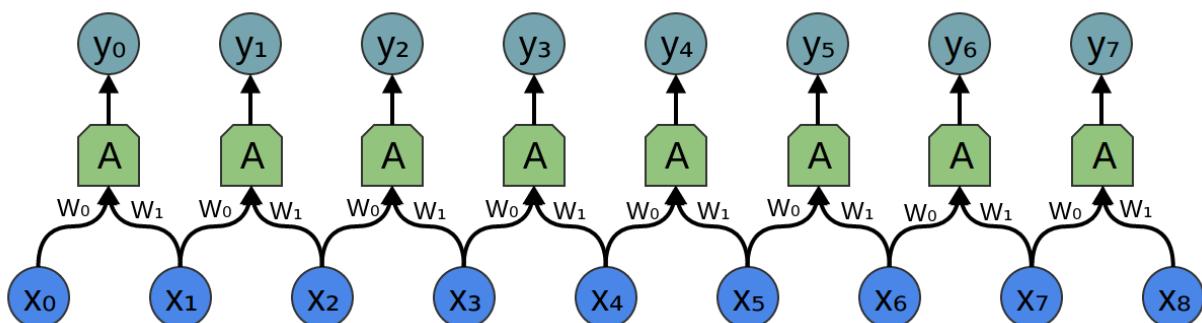
$$(f * g)(c_1, c_2) = \sum_{\substack{a_1 + b_1 = c_1 \\ a_2 + b_2 = c_2}} f(a_1, a_2) \cdot g(b_1, b_2)$$

For CNNs, convolutions allow us to describe which weights are identical across a layer.

A layer of neurons is typically described in aggregate with some weight matrix W , i.e.

$$y = \sigma(Wx + b)$$

In a convolutional layer, many of the weights are repeated in different positions:



A simple CNN [Source](#)

So a typical weight matrix for a layer might look like:

$$W = \begin{bmatrix} W_{0,0} & W_{0,1} & W_{0,2} & W_{0,3} & \dots \\ W_{1,0} & W_{1,1} & W_{1,2} & W_{1,3} & \dots \\ W_{2,0} & W_{2,1} & W_{2,2} & W_{2,3} & \dots \\ W_{3,0} & W_{3,1} & W_{3,2} & W_{3,3} & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

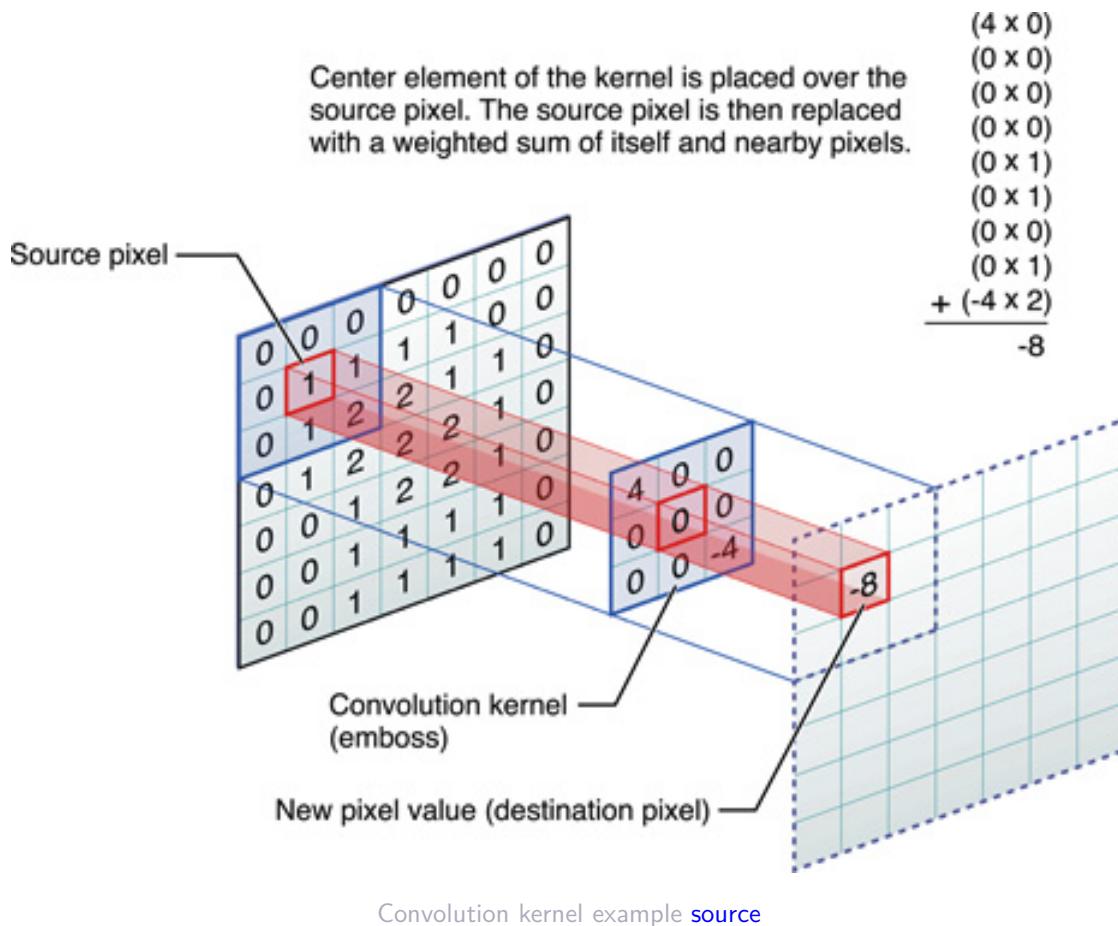
But for a convolutional layer, many weights repeat in different positions. And there are also many zeros since each unit in A is not connected to every input:

$$W = \begin{bmatrix} w_0 & w_1 & 0 & 0 & \dots \\ 0 & w_0 & w_1 & 0 & \dots \\ 0 & 0 & w_0 & w_1 & \dots \\ 0 & 0 & 0 & w_0 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

Multiplying with weight matrix is the same thing as convolving with $[...0, w_1, w_0, 0...]$.

11.12.2 Convolution kernels

CNNs learn a *convolution kernel* and (for images) apply it to every pixel across the image:



11.12.3 References

- <https://colah.github.io/posts/2014-07-Conv-Nets-Modular/>
- <https://colah.github.io/posts/2014-07-Understanding-Convolutions/>
- Composing Music with Recurrent Neural Networks. Daniel Johnson. August 3, 2015. <http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/>

11.13 Recurrent Neural Networks (RNN)

With machine learning, data is typically represented in vector form. This works for certain kinds of data, such as numerical data, but not necessarily for other kinds of data, like text. We usually end up coercing text into some vector representation (e.g. TF-IDF) and end up losing much of its structure (such as the order of words). This is ok for some tasks (such as topic detection), but for many others we are throwing out important information. We could use bigrams or trigrams or so on to preserve some structure but this becomes unmanageably large (we end up with very high-dimension vectors).

Recurrent neural networks are able to take *sequences* as input, i.e. iterate over a sequence, instead of fixed-size vectors, and as such can preserve the sequential structure of things like text and have a stronger concept of “context”.

Basically, an RNN takes in each item in the sequence and updates its hidden representation based on that item and the previous hidden representation. If there is no previous hidden representation (i.e. we are looking at the first item in the sequence), we can initialize it as either all zeros or treat the initial hidden representation as another parameter to be learned.

The input item can be represented with *one-hot encoding*, i.e. each term is to a vector of all zeroes and one 1. For example, if we had the vocabulary the, mad, cat, the terms might be respectively represented as [1, 0, 0], [0, 1, 0], [0, 0, 1].

Another way to represent these terms is with an *embedding matrix*, in which each term is mapped to some index of the matrix which points to some n -dimensional vector representation. So the RNN learns vector representations for each term.

Convolutional neural networks, and feed-forward neural networks in general, treat an input the same no matter when they are given it. For RNNs, the hidden representation is like (short-term) “memory” for the network, so context is taken into account for inputs; that is, an input will be treated differently depending on what the previous input(s) was/were.

RNNs may incorporate **long short term memory** (LSTM) units, which can handle longer-term context. These units have a few gates:

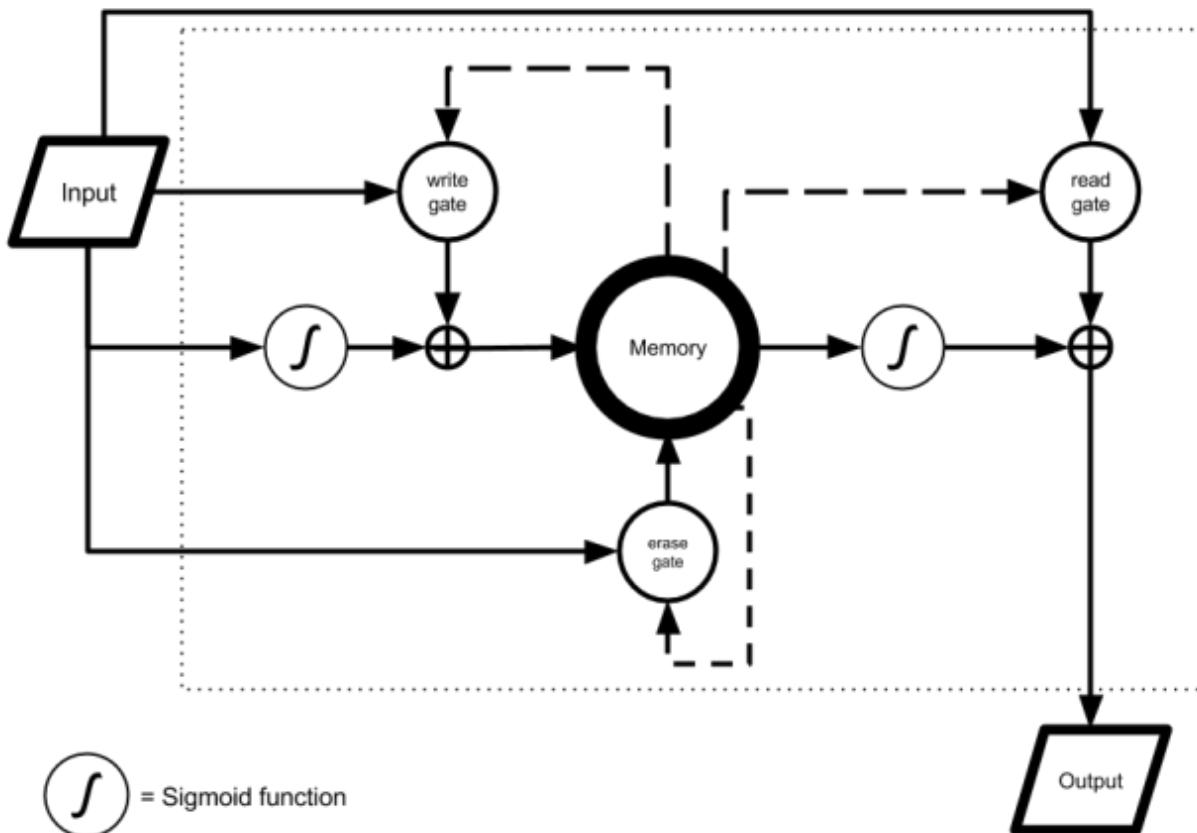
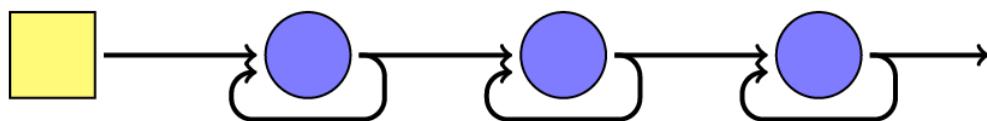
- *write* - controls the amount of current input to be remembered
- *read* - controls the amount of memory given as output to the next stage
- *erase* - controls what part of the memory is erased or kept in the current time step

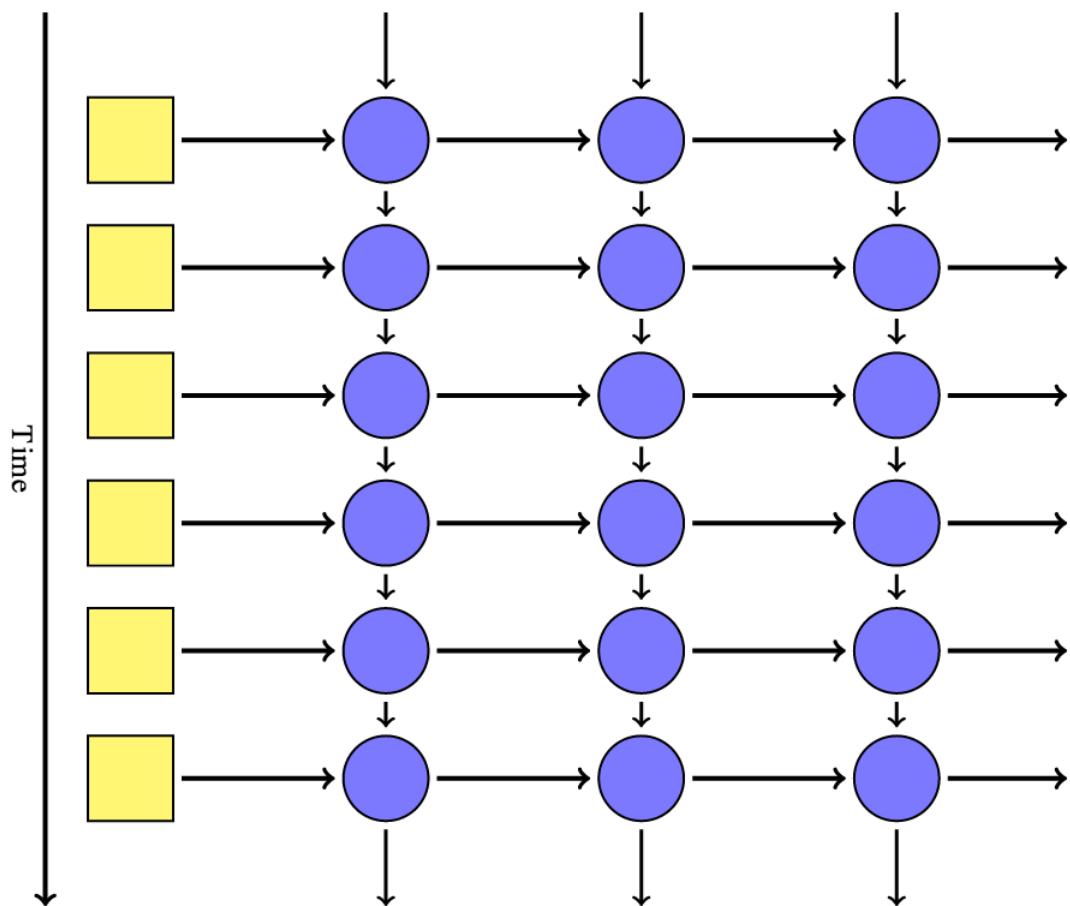
11.13.1 More on RNNs

In the most basic RNN, the hidden layer have two inputs: the input from the previous layer, and the layer’s own output from the previous time step (so it loops back onto itself):

This simple network can be visualized over time as well:

Say we have a hidden layer L_1 of size 3 and another hidden layer L_2 of size 2. In a regular NN, the input to L_2 is of size 3 (because that’s the output size of L_1). In an RNN, L_2 would have 3+2 inputs, 3 from L_1 , and 2 from its own previous output.

A LSTM unit [source](#)Simple RNN network, with hidden nodes looping [source](#)



Simple RNN network, with hidden nodes looping over time [source](#)

This simple feedback mechanism offers a kind of short-term memory - the network “remembers” the output from the previous time step.

It also allows for variable-sized inputs and outputs - the inputs can be fed in one at a time and combined by this feedback mechanism.

This short-term memory may be *too* short, however. Many RNNs incorporate *long short-term memory* (LSTM) instead, where we have memory stored and passed through a longer number of steps. This memory is modified in each step, with something being added and something being removed at each step.

11.13.2 Caveats

- RNNs train very slowly on CPUs; they train significantly faster on GPUs.
- Seems to perform less well than simpler models on small datasets; benefits don’t show up until larger amounts of training data are used

11.13.3 References

- General Sequence Learning using Recurrent Neural Networks, Alec Radford <https://www.youtube.com/watch?v=VINCQghQRuM>
- <http://devblogs.nvidia.com/parallelforall/understanding-natural-language-deep-neural-networks-using-tor>
- Composing Music with Recurrent Neural Networks. Daniel Johnson. August 3, 2015. <http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/>

11.14 Word Embeddings

A word embedding $W : \text{words} \rightarrow \mathbb{R}^n$ is a parameterized function that maps words to high-dimensional vectors (typically 200-500 dimensions).

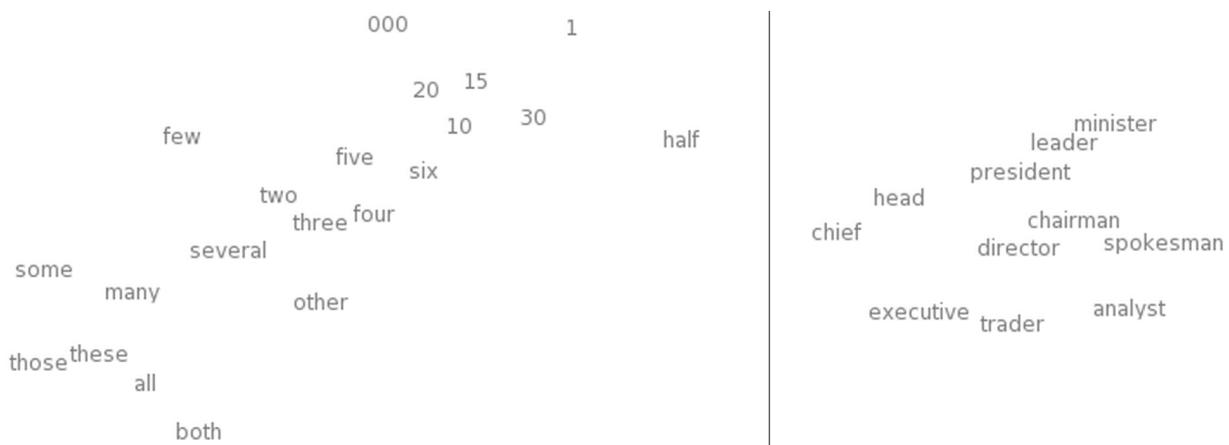
This function is typically a lookup table parameterized by a matrix θ , where each row represents a word. That is, the function is often $W_\theta(w_n) = \theta_n$. θ is initialized with random vectors for each word.

So given a task involving words, we want to learn W so that we have good representations for each word.

You can visualize a word embedding space using t-SNE (a technique for visualizing high-dimensional data):

As you can see, words that are similar in meaning tend to be closer together. Intuitively this makes sense - if words have similar meaning, they are somewhat interchangeable, so we expect that their vectors be similar too.

We’ll also see the vectors capture notions of analogy, for example “Paris” is to “France” as “Tokyo” is to “Japan”. These kinds of analogies can be represented as vector addition: “Paris” - “France” + “Japan” = “Tokyo”.

Visualizing a word embedding space with t-SNE ([Turian et al \(2010\)](#))

The best part is the neural network is not explicitly told to learn representations with these properties - it is just a side effect. This is one of the remarkable properties of neural networks - they learn good ways of representing the data more or less on their own.

And these representations can be portable. That is, maybe you learn W for one natural language task, but you may be able to re-use W for another natural language task (provided it's using a similar vocabulary). This practice is sometimes called "pretraining" or "transfer learning" or "multi-task learning".

You can also map multiple words to a single representation, e.g. if you are doing a multilingual task. For example, the English and French words for "dog" could map to the same representation since they mean the same thing (in which case we could call this a "bilingual word embedding").

Here's an example visualization of a Chinese and English bilingual word embedding:

You can even go a step further and learn image and word representations together, so that vectors representing images of horses are close to the vector for the word "horse".

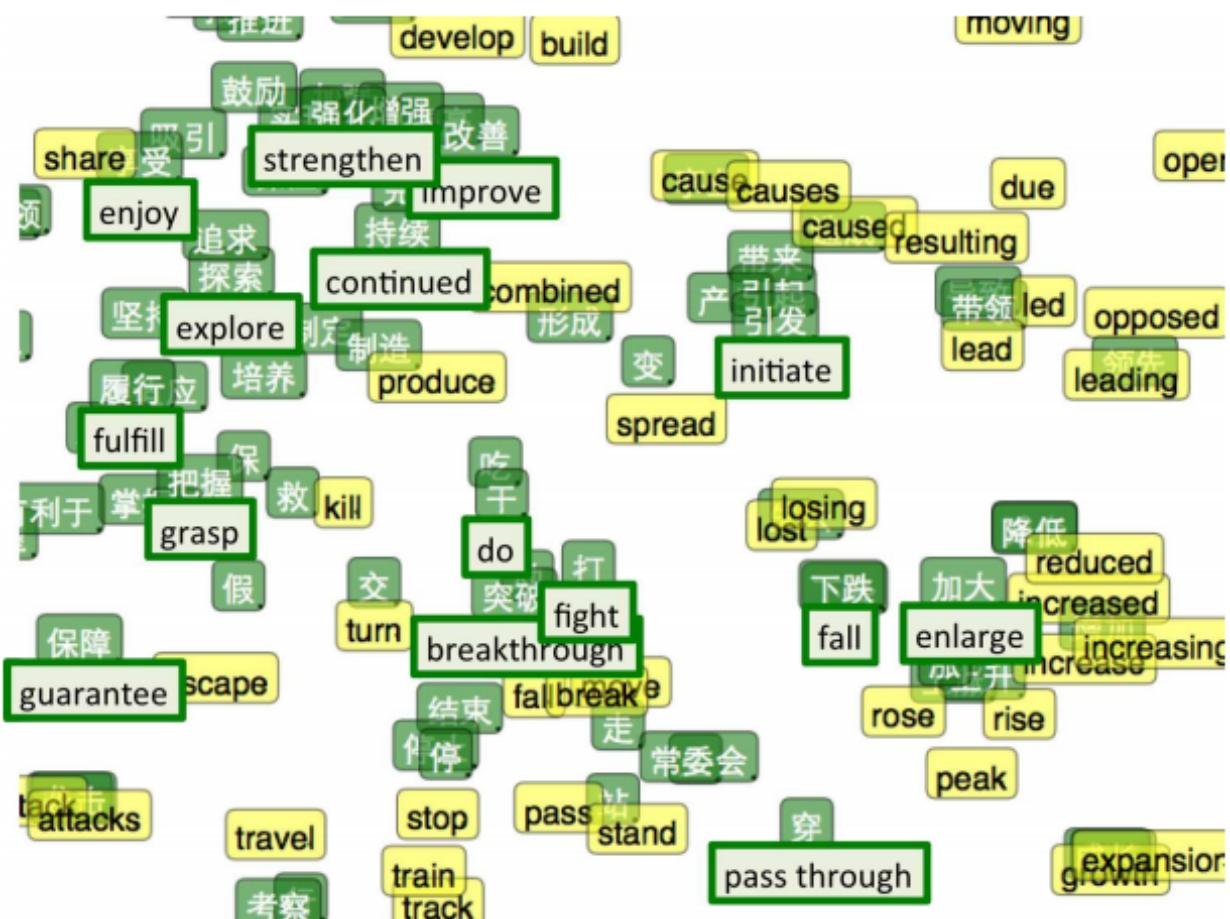
Two main techniques for learning word embeddings are:

- CBOW: predicting the probability of context words given a word
- Skip-gram: predicting the probability of a word given context words

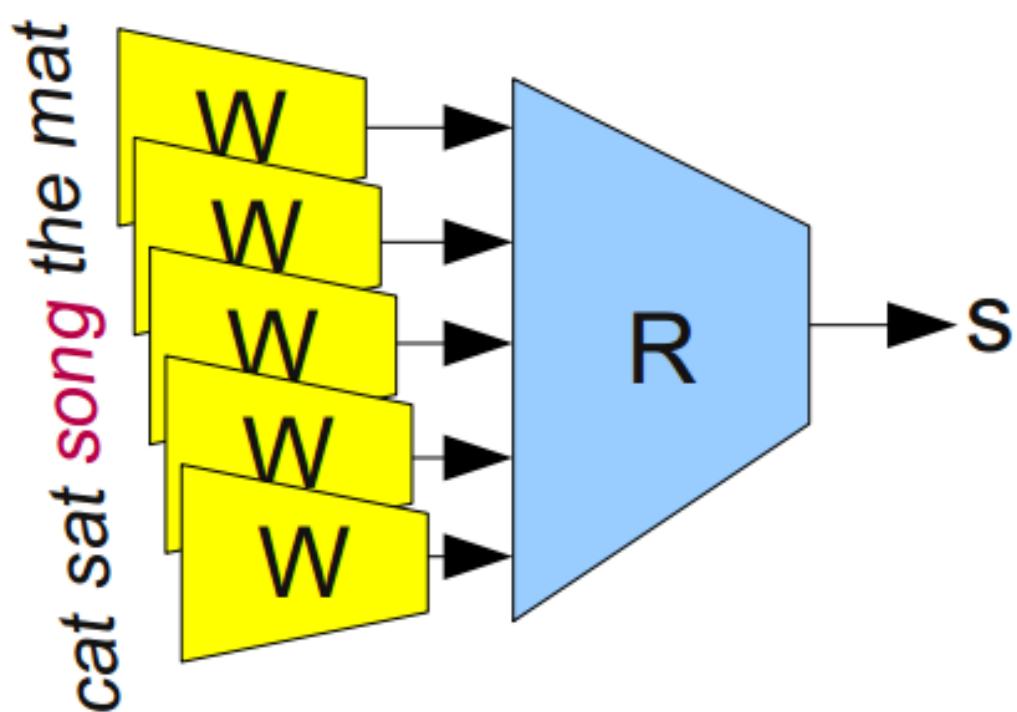
11.14.1 Modular Neural Networks

So say we have trained a neural net which has learned our function W , and given a word input, it outputs us the word's high-dimensional vector representation.

We can re-use this network in a modular fashion so that we construct a larger neural net which can take a fixed-size set of words as input. For example, the following network takes in five words, from which we get their representations, which are then passed into another network R to yield some output s .



A Chinese and English word embedding ([Socher et al \(2013a\)](#))

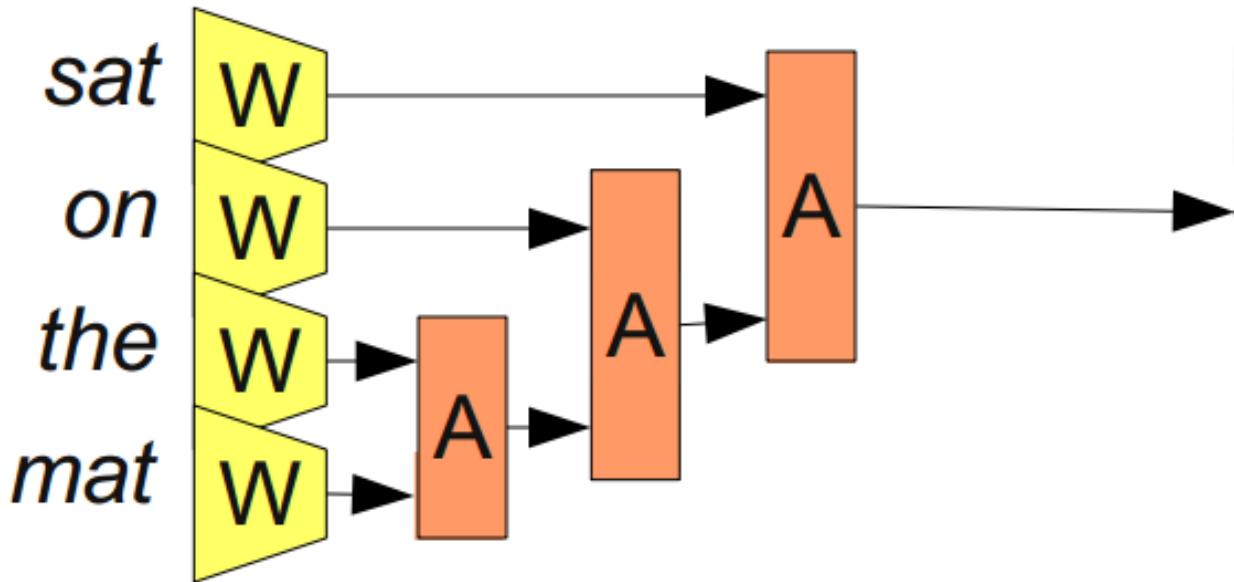


A modular neural network ([Bottou \(2011\)](#))

11.14.2 Recursive Neural Networks

Using modular neural networks like above is limiting in the fact that we can only accept a fixed number of inputs.

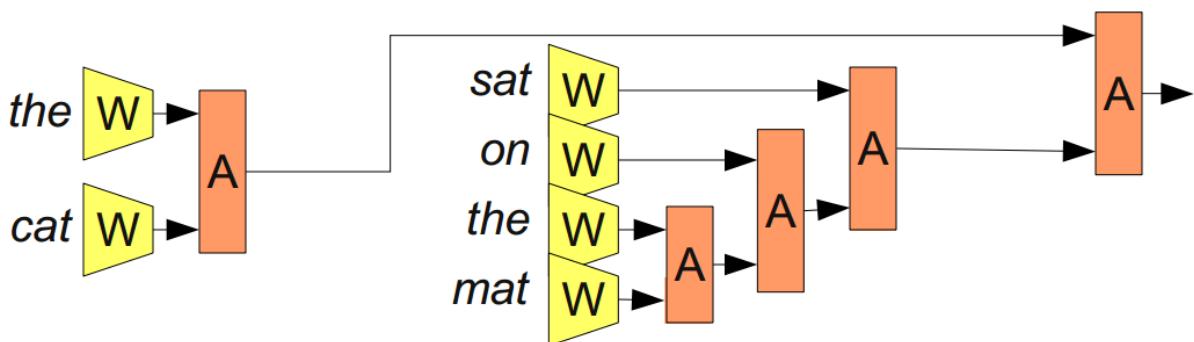
We can get around this by adding an association module A , which takes two representations and merges them.



Using association modules ([Bottou \(2011\)](#))

As you can see, it can take either a reputation from a word (via a W module) or from a phrase (via another A module).

We probably don't want to merge words linearly though. Instead we might want to group words in some way:



A recursive neural network ([Bottou \(2011\)](#))

This kind of model is a “recursive neural network” (sometimes “tree-structured neural network”) because it has modules feeding into modules of the same type.

11.15 Nonlinear neural nets

In typical NNs, the architecture of the network is specified before hand and is static - neurons don't change connections. In a nonlinear neural net, however, the connections between neurons becomes dynamic, so that new connections may form and old connections may break. This is more like how the human brain operates. But so far at least, these are very complex and difficult to train.

11.16 Transfer learning

The practice of transfer learning involves taking a neural net trained for another task and applying it to a different task. For instance, if using an image classification net trained for one classification task, you can use that same network for another, truncating the output layer, that is, take the vectors from the second-to-last layer and use those as feature vectors for other tasks.

11.17 Recursive Neural Tensor Networks

~ to do ~

11.17.1 References

- <https://colah.github.io/posts/2014-07-NLP-RNNs-Representations/>
- <https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-neural-network>
 - i- <http://www.faqs.org/faqs/ai-faq/neural-nets/part1/preamble.html>

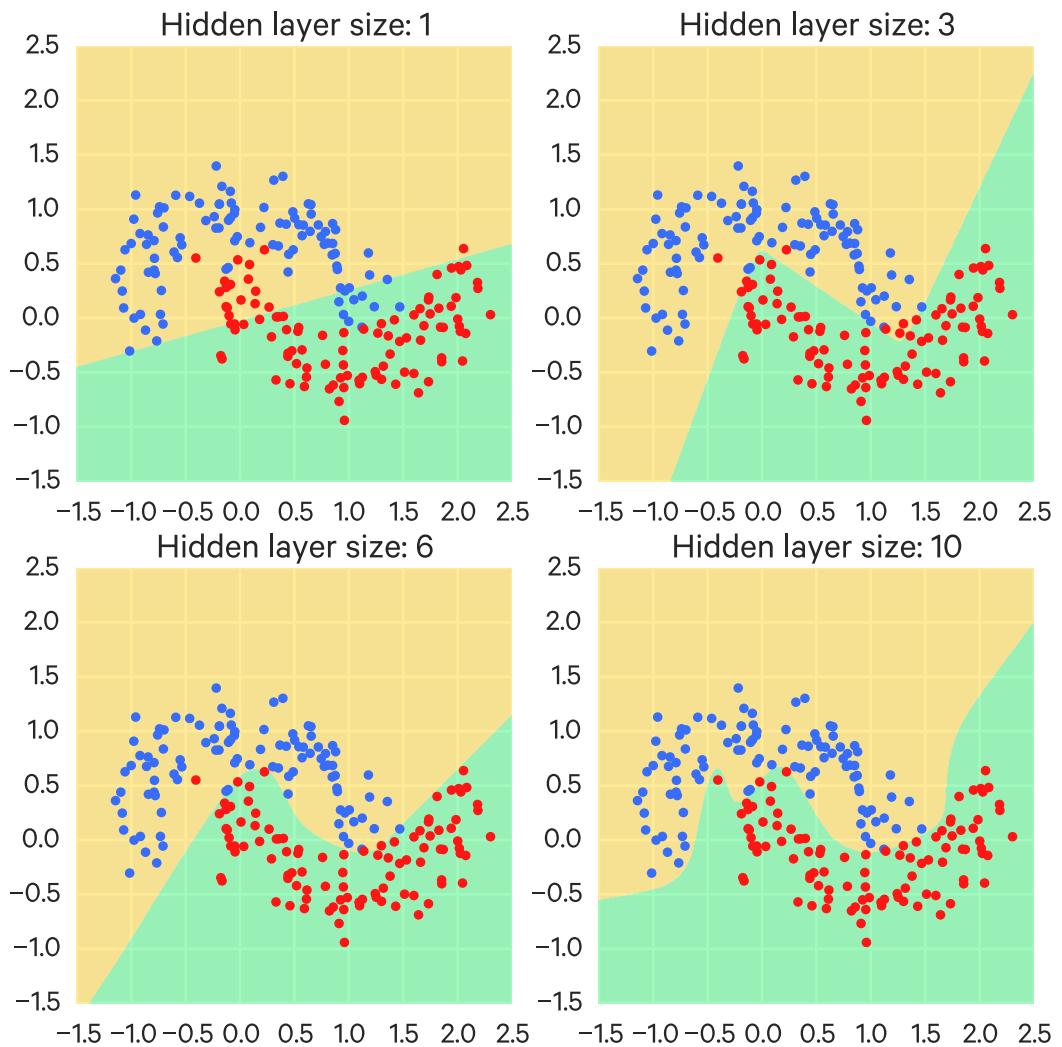
11.18 Choosing the network architecture

How do you decide how many layers to use, and what size each layer should be?

As the network grows in number of layers and size, the network *capacity* increases, which is to say it is capable of representing more complex functions.

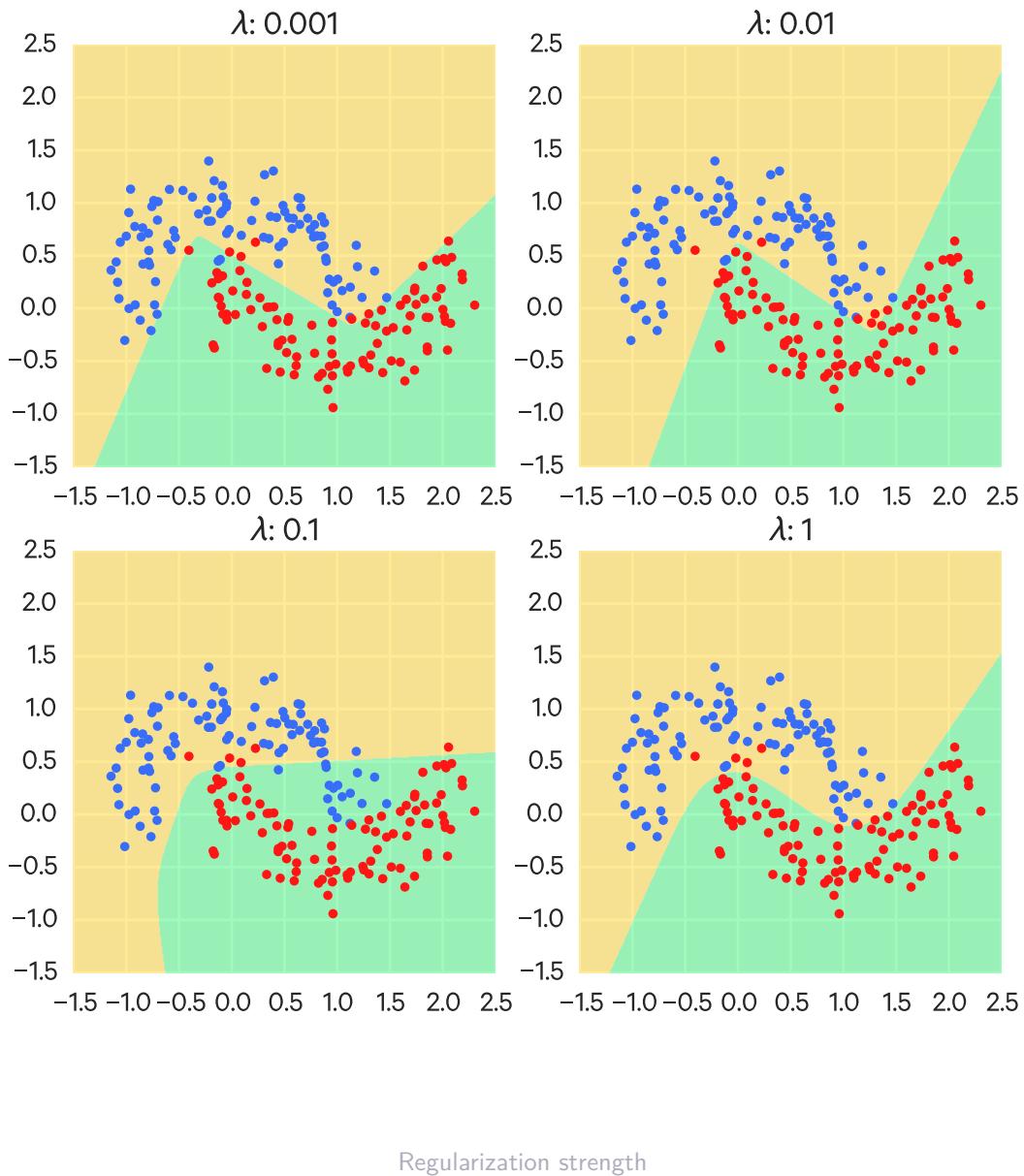
Simpler networks have fewer local minima, but they are easier to converge to and tend to perform worse (they have higher loss). There is a great deal of variance across these local minima, so the outcome is quite sensitive to the random initialization - some times you land in a good local minima, sometimes not. More complex networks have more local minima, but they tend to perform better, and there is less variance across how these local minima perform.

Higher-capacity networks run a greater risk of overfitting, but this overfitting can be (preferably) mitigated by other methods such as L2 regularization, dropout, and input noise. So don't let overfitting be the sole reason for going with a simpler network if a larger one seems appropriate.



More complex network, more complex functions

Here are regularization examples for the same data from the previous image, with the neural net for 20 hidden neurons:



Regularization strength

As you can see, regularization is effective at counteracting overfitting.

11.18.1 References

- CS231n Convolutional Neural Networks for Visual Recognition, Module 1: Neural Networks Part 1: Setting up the Architecture. Andrej Karpathy. <https://cs231n.github.io/neural-networks-1/>

11.19 Weight initialization

Given normalized data, we estimate that roughly half the weights will be negative and roughly half will be positive.

It may seem intuitive to initialize all weights to zero, but you should not, since this causes every neuron to have the same output, which causes them to have the same gradients during backpropagation, which causes them to all have the same parameter updates. Thus none of the neurons will differentiate.

So we can set each neuron's initial weights to be a random vector from a standard multidimensional normal distribution, scaled by some value, e.g. 0.001 so that they are kept very small, but still non-zero. This process is known as *symmetry breaking*. The random initializations allow the neurons to differentiate themselves during training.

Note that, however, small initial weights can be problematic for deep networks, since they may reduce the gradient signal that flows backwards by too much (in a weaker version of the gradient "killing" effect mentioned earlier).

As the number of inputs to a neuron grows, so too will its output's variance. This can be controlled for (calibrated) by scaling its weight vector by the square root of its "fan-in" (its number of inputs), so you should divide the standard multidimensional distribution sampled random vector by \sqrt{n} , where n is the number of the neuron's inputs. For ReLUs, it is recommended you instead divide by $\sqrt{2/n}$. ([Karpathy's CS231n notes](#) provides more detail on why this is.)

An alternative to this fan-in scaling for the uncalibrated variances problem is *sparse initialization*, which is to set all weights to 0, and then break symmetry by randomly connecting every neuron to some fixed number (e.g. 10) of neurons below it by setting those weights to ones randomly sampled from the standard normal distribution like mentioned previously.

Biases are commonly initialized to be zero, though if using ReLUs, then you can set them to a small value like 0.01 so all the ReLUs fire at the start and are included in the gradient backpropagation update.

11.19.1 References

- CS231n Convolutional Neural Networks for Visual Recognition, Module 2: Neural Networks Part 2: Setting up the Data and the Loss. Andrej Karpathy. <https://cs231n.github.io/neural-networks-2/>

11.20 Regularization

Regularization techniques are used to prevent neural networks from overfitting.

11.20.1 L2 Regularization

The most common form of regularization. Penalize the squared magnitude of all parameters (weights) as part of the objective function, i.e. we add $\sum \lambda w^2$ to the objective function. It is common to include $\frac{1}{2}$, i.e. use $\frac{1}{2} \sum \lambda w^2$, so the gradient of this term wrt to w is just λw instead of $2\lambda w$. This avoids the network relying heavily on a few weights and encourages it to use all weights a little.

11.20.2 L1 Regularization

Similar to L2 regularization, except that the term added to the objective function is $\sum \lambda |w|$. L1 regularization has the effect of causing weight vectors to become sparse, such that neurons only use a few of their inputs and ignore the rest as “noise”. Generally L2 regularization is preferred to L1.

11.20.3 Elastic net regularization

This is just the combination of L1 and L2 regularization, such that the term introduced to the objective function is $\sum \lambda_1 |w| + \lambda_2 w^2$.

11.20.4 Max norm constraints

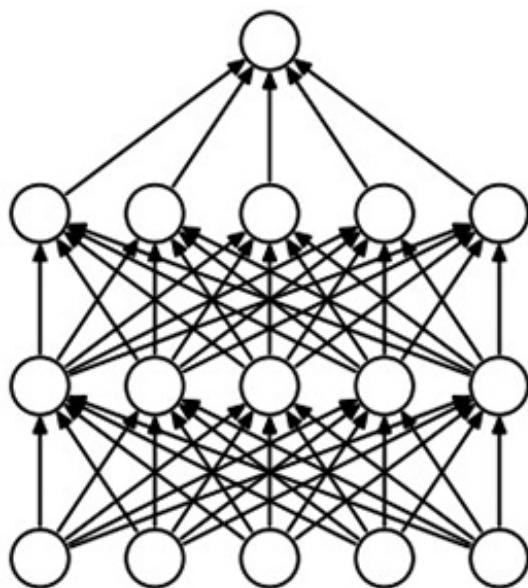
This involves setting an absolute upper bound on the magnitude of the weight vectors; that is, after updating the parameters/weights, clamp every weight vector so that it satisfies $\|w\|_2 < c$, where c is some constant (the maximum magnitude).

11.20.5 Dropout

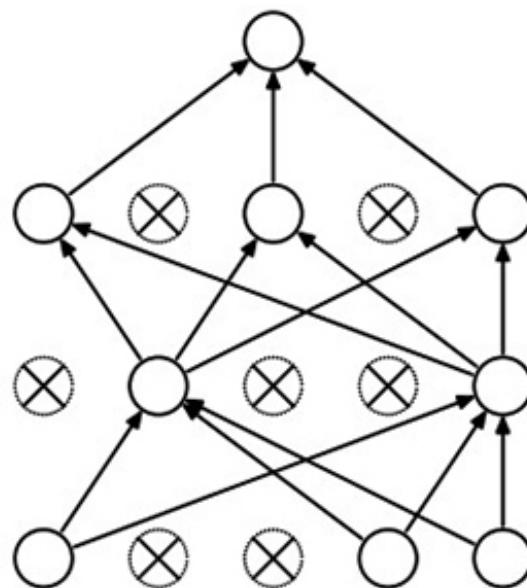
Dropout is a regularization method which works well with the others mentioned so far (L1, L2, maxnorm). During training, we specify a probability p , and we only keep a neuron active with that probability p , otherwise we set its output to zero. If the neuron’s output is set to 0, that has the effect of temporarily “removing” that neuron for that training iteration. This dropout is applied only at training time and applied per-layer (that is, it is applied after each layer, see the code example below). This prevents the network from relying too much on certain neurons.

One way to think about this is that, for each training step, a sub-network is sampled from the full network, and only those parameters are updated. Then on the next step, a different sub-sample is taken and updated, and so on.

At test time, all neurons are active (i.e. we don’t use dropout at test time). However, we must scale the activation functions by p to maintain the same expected output for each neuron. Say x is the output of a neuron without dropout. With dropout, the neuron’s output has a chance p of being set to 0, so its expected output becomes px (more verbosely, it has $1 - p$ chance of becoming 0, so its output is $px + (1 - p)0$, which simplifies to px). Thus we must scale the outputs (i.e. the activation functions) by p to keep the expected output consistent.



(a) Standard Neural Net



(b) After applying dropout.

A network after dropout is applied to each layer in a training iteration [source](#)

This scaling can be applied at training time, which is more efficient - this technique is called *inverted dropout*.

For comparison, here is an implementation of regular dropout and an implementation of inverted dropout (source from: <https://cs231n.github.io/neural-networks-2/>)

```
# Dropout
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
```

```

out = np.dot(W3, H2) + b3

# Inverted dropout
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3

```

11.20.6 Recommendations

It is most common to use a single, global L2 regularization strength that is cross-validated. It is also common to combine this with dropout applied after all layers. The value of $p = 0.5$ is a reasonable default, but this can be tuned on validation data. <https://cs231n.github.io/neural-networks-2/>

11.20.7 References

- CS231n Convolutional Neural Networks for Visual Recognition, Module 2: Neural Networks Part 2: Setting up the Data and the Loss. Andrej Karpathy. <https://cs231n.github.io/neural-networks-2/>

11.21 Training

Start training with small, unequal weights to avoid *saturating* the network w/ large weights. If all the weights start equal, the network won't learn anything.

11.22 Hopfield Nets

11.22.1 Recurrent NNs and stability

RNNs typically work by feeding in input, taking the output that results and feeding it in as new input, and so on until the output stabilizes. It is possible, however, that the output never stabilizes; such an RNN is said to be *unstable*.

In a binary Hopfield net, a simple threshold function is used as the activation function, such that each neuron outputs 0 or 1. The *state* of the network is the set of values the network outputs, which effectively is a binary number, e.g. 1001 is a possible state for a network with four outputs. These outputs are fed back into the network as inputs which may change the state of the network - the hope is that eventually it stabilizes.

A Hopfield net with two outputs has four possible states: 00, 01, 10, 11, which can be conceptualized as the vertices of a two-dimensional polygon (a square). More generally, a Hopfield net with n outputs will have 2^n states, and can be represented by an n -dimensional hypercube.

If we represent the weights of the network as a matrix W , then we can prove that the recurrent network is stable given that $w_{ii} = 0$ for all i and that $w_{ij} = w_{ji}$ for all $j \neq i$; that is, if the matrix is symmetrical and as a zero diagonal. (see pp97) This can be similarly proven for continuous recurrent neural networks (i.e. with continuous activation functions).

However, it is possible for a recurrent network that does not satisfy these properties to be stable (which is to say, these are sufficient but not necessary requirements).

Note that a stable Hopfield net does not necessarily stabilize on the global minimum. Here, statistical training methods such as Boltzmann or Cauchy training can be used to guarantee a global minimum.

For the binary Hopfield net, the following strategy can be used:

$$E_k = \text{NET}_k - \theta_k$$

$$p_k = \frac{1}{1 + \exp(-\delta \frac{E_k}{T})}$$

where:

- NET_k = the NET output of neuron k
- θ_k = the threshold of neuron k
- T = the artificial temperature

Set T to a high value and set the neurons to an initial state by passing in an input vector. Then repeat:

1. For each neuron, set the state to one with probability p_k , otherwise, set to zero
2. Gradually reduce the artificial temperature and repeat step 1 until equilibrium is reached

backprop:

a single neuron:

$$\text{OUT} = F(\text{NET})$$

where:

- $\text{NET} = XW$ = net input to the activation function
- X = input vector
- W = weight vector
- F = activation function
- OUT = output of the neuron

if F is the sigmoid function:

$$\frac{\partial \text{OUT}}{\partial \text{NEXT}} = (\text{OUT})(1 - \text{OUT})$$

The total error of the network is: $\text{ERROR}_{\text{total}} = y_{\text{true}} - y_{\text{pred}}$.

We'll call the error for a particular layer ERROR .

For the penultimate layer (the layer just before the output), $\text{ERROR} = \text{ERROR}_{\text{total}}$.

For every preceding layer:

$$\text{ERROR} = W_{L+1}\delta_{L+1}$$

That is, it is equal to the product of the weight (before updating it) and error for the layer that comes after it (forward-wise).

The error for a particular neuron is:

$$\delta = \frac{\partial \text{OUT}}{\partial \text{NEXT}}(\text{ERROR})$$

The weight update for a particular neuron is:

$$\Delta W = \delta \eta \text{OUT}$$

So the new weight is just:

$$W+ = \Delta W$$

η is the learning rate - it is preferable to α for notation because α can be used to represent a “momentum coefficient” or a “smoothing coefficient”.

Momentum: keep track of the previous weight change and incorporate it in the next weight updates, such that:

$$\Delta = \eta \delta \text{OUT} + \alpha [\Delta W_{t-1}]$$

Where $\alpha \in [0, 1]$ is the *momentum coefficient*, usually 0.9, and the final bracketed term, ΔW_{t-1} , is the weight change from the previous update.

But most often, momentum is not helpful.

Exponential smoothing:

$$\Delta W = \eta(\alpha[\Delta W_{t-1}] + (1 - \alpha)\delta \text{OUT})$$

where $\alpha \in [0, 1]$. Exponential smoothing can be better than momentum.

Using the sigmoid activation function is problematic because the outputs (OUT) have a good chance of being around 0, in which case the weight is not updated, because $\Delta W = \delta \eta \text{OUT}$, that is, if $\text{OUT} \approx 0$, then $\Delta W \approx 0$. Thus the network does not learn (the weights don't change). You can continue to use the sigmoid function if you adjust it:

$$\text{OUT} = -\frac{1}{2} + \frac{1}{e^{-\text{NET}} + 1}$$

but, as written elsewhere, you should just use a different activation function, like tanh.

backprop is *not* a guarantee of training at all nor of quick training.

possible issues:

- network paralysis: if the weights become very large, the neurons' OUTs may become very large, where the derivative of the activation function is very small, so weights are not really updated and get “stuck” at large values.

- local minima: statistical training methods can be used (such as simulated annealing), but increase training time
 - step size: if it is too small, training is too slow, if it is too large, paralysis or instability (no convergence) are possible
 - stability: that the network does not mess up its learning of something else to learn another thing. for instance, say it learns good weights for one input, but to learn good weights for another input, it “overwrites” or “forgets” what it learned about the prior input.
-

statistical training methods:

statistical (or “stochastic”) training methods, contrasted with *deterministic* training methods, involve some randomness to avoid local minima. They generally work by randomly leaving local minima to possibly find the global minimum. The severity of this randomness decreases over time so that a solution is “settled” upon (this gradual “cooling” of the randomness is the key part of *simulated annealing*).

simulated annealing applied as a training method to a neural network is called *Boltzmann training* (neural networks trained in this way are called *Boltzmann machines*):

1. set T (the artificial temperature) to a large value
2. apply inputs, calculate outputs and objective function
3. make random weight changes, recalculate network output and change in objective function
 - 4a. if objective function improves, keep weight changes
 - 4b. if the objective function worsens, accept the change according to the probability drawn from Boltzmann distribution, $P(c)$, select a random variable r from a uniform distribution in $[0, 1]$; if $P(c) > r$, keep the change, otherwise, don't.

$$P(c) = \exp\left(\frac{-c}{kT}\right)$$

where:

- c the change in the objective function
- k a constant analogous to the Boltzmann's constant in simulated annealing, specific for the current problem
- T the artificial temperature
- $P(c)$ the probability of the change c in the objective function

Steps 3 and 4 are repeated for each of the weights in the network as T is gradually decreased.

The random weight change can be selected in a few ways, but one is just choosing it from a Gaussian distribution, $P(w) = \exp\left(\frac{-w^2}{T^2}\right)$, where $P(w)$ is the probability of a weight change of size w and T

is the artificial temperature. Then you can use Monte Carlo simulation to generate the actual weight change, Δw .

Boltzmann training uses the following cooling rate, which is necessary for convergence to a global minimum:

$$T(t) = \frac{T_0}{\log(1+t)}$$

Where T_0 is the initial temperature, and t is the artificial time.

The problem with Boltzmann training is that it can be very slow (the cooling rate as computed above is very low).

This can be resolved by using the Cauchy distribution instead of the Boltzmann distribution; the former has fatter tails so has a higher probability of selecting large step sizes. Thus the cooling rate can be much quicker:

$$T(t) = \frac{T_0}{1+t}$$

The Cauchy distribution is:

$$P(x) = \frac{T(t)}{T(t)^2 + x^2}$$

where $P(x)$ is the probability of a step of size x .

This can be integrated, which makes selecting random weights much easier:

$$x_c = \rho T(t) \tan(P(x))$$

Where ρ is the learning rate coefficient and x_c is the weight change.

Here we can just select a random number from a uniform distribution in $(-\frac{\pi}{2}, \frac{\pi}{2})$, then substitute this for $P(x)$ and solve for x in the above, using the current temperature.

Cauchy training still may be slow so we can also use a method based on *artificial specific heat* (in annealing, there are discrete energy levels where phase changes occur, at which abrupt changes in the “specific heat” occur). In the context of artificial neural networks, we define the (pseudo)specific heat to be the average rate of change of temperature with the objective function. The idea is that there are parts where the objective function is sensitive to small changes in temperature, where the average value of the objective function makes an abrupt change, so the temperature must be changed slowly here so as not to get stuck in a local minima. Where the average value of the objective function changes little with temperature, large changes in temperature can be used to quicken things.

Still, Cauchy training may be much slower than backprop, and can have issues of network paralysis (because it is possible to have very large random weight changes), esp. if a nonlinearity is used as the activation function (see the bit on network paralysis and the sigmoid function above).

Cauchy training may be combined with backprop to get the best of both worlds - it simply involves computing both the backprop and Cauchy weight updates and applying their weighted sum as the update. Then, the objective function's change is computed, and like with Cauchy training, if there is an improvement, the weight change is kept, otherwise, it is kept with a probability determined by the Boltzmann distribution.

The weighted sum of the individual weight updates is controlled by a coefficient η , such that the sum is $\eta[\alpha\Delta W_{t-1} + (1 - \alpha)\delta \text{OUT}] + (1 - \eta)x_c$, so that if $\eta = 0$, the training is purely Cauchy, and if $\eta = 1$, it becomes purely backprop.

There is still the issue of the possibility of retaining a massive weight change due to the Cauchy distribution's infinite variance, which creates the possibility of network paralysis. The recommended approach here is to detect saturated neurons by looking at their OUT values - if it is approaching the saturation point (positive or negative), apply some squashing function to its weights (note that this squashing function is not restricted to the range $[-1, 1]$ and in fact may work better with a larger range). This potently reduces large weights while only attenuating smaller ones, and maintains symmetry across weights.

11.23 Unsupervised neural networks

The most basic one is probably the autoencoder, which is a feed-forward neural net which tries to predict its own input. While this isn't exactly the world's hardest prediction task, one makes it hard by somehow constraining the network. Often, this is done by introducing a bottleneck, where one or more of the hidden layers has much lower dimensionality than the inputs. Alternatively, one can constrain the hidden layer activations to be sparse (i.e. each unit activates only rarely), or feed the network corrupted versions of its inputs and make it reconstruct the clean ones (this is known as a denoising autoencoder).
[\[https://www.metacademy.org/roadmaps/rgrosse/deep_learning\]](https://www.metacademy.org/roadmaps/rgrosse/deep_learning)

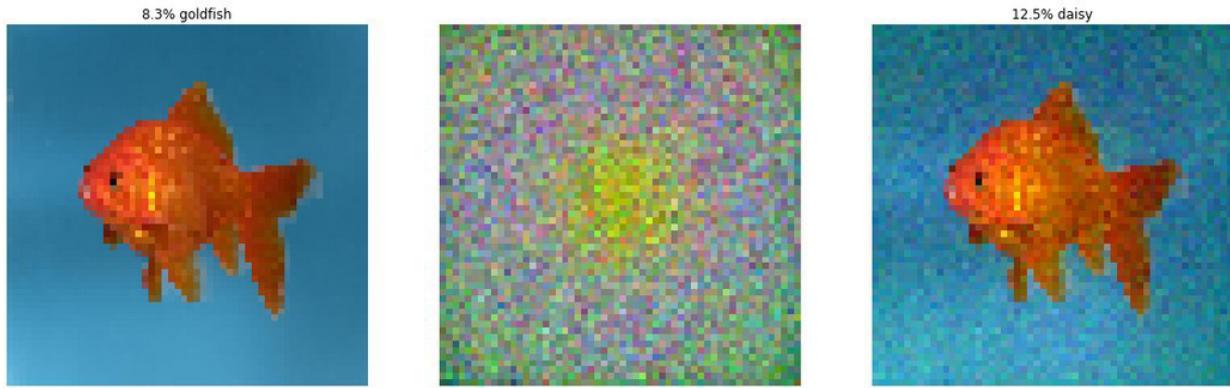
11.24 In high dimensions, local minima are not problems

More typically, there are saddle points, which slow down training but can be escaped in time. This is because with many dimensions, it is unlikely that a point is a minimum in *all* dimensions (if we consider that a point is a minimum in one dimensions with probability p , then it has probability p^n to be a minimum in all n dimensions); it is, however, likely that it is a local minimum in some of the dimensions.

As training nears the global minimum, p increases, so if you do end up at a local minimum, it will likely be close enough to the global minimum.

11.25 Training on adversarial examples

Adding noise to input, such as in the accompanying figure, can throw off a classifier. Few strategies are robust against these tricks, but one approach is to generate these adversarial examples and include them as part of the training set.



11.26 Training neural nets tips

- Normalize real-valued data (subtract mean, divide by standard deviation (see part on data preprocessing))
- Decrease the learning rate during training
- Use minibatches for a more stable gradient
- Use momentum to get through plateaus

11.27 Neural net weight initialization

Sample your weights uniformly from $[-b, b]$, where:

$$b = \sqrt{\frac{6}{H_k + H_{k+1}}}$$

where H_k and H_{k+1} are the sizes of the hidden layers before and after the weight matrix.

11.28 Activation functions

| Activation Function | Propagation | Backpropagation |
|---------------------|--------------------------------|---|
| Sigmoid | $y_s = \frac{1}{1+e^{-x_2}}$ | $[\frac{\partial E}{\partial x}]_s = [\frac{\partial E}{\partial y}]_s \frac{1}{(1+e^{x_2})(1+e^{-x_2})}$ |
| Tanh | $y_s = \tanh(x_s)$ | $[\frac{\partial E}{\partial x}]_s = [\frac{\partial E}{\partial y}]_s \frac{1}{\cosh^2 x_s}$ |
| ReLU | $y_s = \max(0, x_s)$ | $[\frac{\partial E}{\partial x}]_s = [\frac{\partial E}{\partial y}]_s \mathbb{I}\{x_s > 0\}$ |
| Ramp | $y_s = \min(-1, \max(1, x_s))$ | $[\frac{\partial E}{\partial x}]_s = [\frac{\partial E}{\partial y}]_s \mathbb{I}\{1 - < x_s < 1\}$ |

11.29 Loss functions

| Loss Function | Propagation | Backpropagation |
|-----------------------------|---|---|
| Square | $y = \frac{1}{2}(x - d)^2$ | $\frac{\partial E}{\partial x} = (x - d)^T \frac{\partial E}{\partial y}$ |
| Log, $c = \pm 1$ | $y = \log(1 + e^{-cx})$ | $\frac{\partial E}{\partial x} = \frac{-c}{1+e^{cx}} \frac{\partial E}{\partial y}$ |
| Hinge, $c = \pm 1$ | $y = \max(0, m - cx)$ | $\frac{\partial E}{\partial x} = -c \mathbb{I}\{cx < m\} \frac{\partial E}{\partial y}$ |
| LogSoftMax, $c = 1 \dots k$ | $y = \log(\sum_k e^{x_k}) - x_c$ | $[\frac{\partial E}{\partial x}]_s = (\frac{e^{x_s}}{\sum_k} e^{x_k} - \delta_{sc}) \frac{\partial E}{\partial y}$ |
| MaxMargin, $c = 1 \dots k$ | $y = [\max_{k \neq c} \{x_k + m\} - x_c]_+$ | $[\frac{\partial E}{\partial x}]_s = (\delta_{sk^*} - \delta_{sc}) \mathbb{I}\{E > 0\} \frac{\partial E}{\partial y}$ |

11.30 LSTM networks

A type of RNN.

Conventional RNNs do not support learning long-term dependencies; LSTMs are a variation on the conventional RNN which includes a mechanism for learning long-term dependencies. Generally, LSTM networks perform better than regular RNNs.

The conventional RNN has a single activation function which processes its input. An LSTM network instead has four interacting layers in the activation function's place. These layers are themselves learned neural network layers. This entire LSTM unit is sometimes called an LSTM cell.

The cell has a *state* which is modified based on *gates*, which manage what information modifies the state. The gates are composed of a sigmoid neural net layer and a pointwise multiplication operation. The output of this layer (which, due to the sigmoid function, is in $[0, 1]$) is the amount of information passed through to modify the state. So it can be nothing (0), all of it (1) or somewhere in between.

Three of the cell's four neural network layers are these sigmoid gates.

One gate is the *forget gate* (sometimes called an *erase gate*), which controls what is removed ("forgotten") from the cell state. The input to the forget gate is the concatenation of the cell's output from the previous step, OUT_{t-1} and the current input to the cell, X_t . The gate computes a value in $[0, 1]$ (with the sigmoid function) for each value in the previous cell state C_{t-1} ; the resulting value determines how much of that value to keep (1 means keep it all, 0 means forget all of it). So

we are left with a vector of values in $[0, 1]$, which we then pointwise multiply with the existing cell state to get the updated cell state.

The output of a forget gate f at step t is:

$$f_t = \text{sigmoid}(W_f[\text{OUT}_{t-1}, X_t] + b_f)$$

Then our intermediate value of C_t is $C'_t = f_t C_{t-1}$.

Where W_f, b_f are the forget gate's weight vector and bias, respectively.

One of the other gates is the *input gate* (sometimes called a *write gate*), which controls what information gets stored in the cell state. This gate also takes as input the concatenation of OUT_{t-1} and X_t . We will denote its output at step t as i_t . Like the forget gate, this is a vector of values in $[0, 1]$ which determine how much information gets through - 0 means none, 1 means all of it.

A tanh function takes the same input and outputs a vector of candidate values, \tilde{C}_t .

We pointwise multiple this candidate value vector with the input gate's output vector to get the vector that is passed to the cell state. This resulting vector is pointwise added to the updated cell state.

$$\begin{aligned} i_t &= \text{sigmoid}(W_i[\text{OUT}_{t-1}, X_t] + b_i) \\ \tilde{C}_t &= \tanh(W_C[\text{OUT}_{t-1}, X_t] + b_C) \end{aligned}$$

Thus our final updated value of C_t is $C_t = C'_t + i_t \tilde{C}_t$.

We don't output this cell state C_t directly. Rather, we have yet another gate, the *output gate* (sometimes called a *read gate*) that outputs another vector with values in $[0, 1]$, o_t , which determines how much of the cell state is outputted. This gate again takes in as input the concatenation of OUT_{t-1} and X_t .

So the output of the output gate is just:

$$o_t = \text{sigmoid}(W_o[\text{OUT}_{t-1}, X_t] + b_o)$$

To get the final output of the cell, we pass the cell state C_t through tanh and then pointwise multiply that with the output of the output gate:

$$\text{OUT}_t = o_t \tanh(C_t)$$

11.30.1 Variations on LSTM

Peephole connections

This variant just passes on the previous cell state, C_{t-1} , to the forget and input gates, and the new cell state, C_t , to the output gate, that is, all that is changed is that:

$$\begin{aligned} f_t &= \text{sigmoid}(W_f[C_{t-1}, \text{OUT}_{t-1}, X_t] + b_f) \\ i_t &= \text{sigmoid}(W_i[C_{t-1}, \text{OUT}_{t-1}, X_t] + b_i) \\ o_t &= \text{sigmoid}(W_o[C_{t-1}, \text{OUT}_{t-1}, X_t] + b_o) \end{aligned}$$

Update gates

In this variant, the forget and input gates are combined into a single *update gate*. The value f_t is computed the same, but i_t is instead just:

$$i_t = 1 - f_t$$

Essentially, we just update enough information to replace what was forgotten.

Gated Recurrent Unit (GRU)

This is a popular variant, which also combines the forget and input gates into an update gate (denoted as r here) and replaces the output gate with another gate, denoted z . The cell state and its output are also merged as its hidden state, h_t . This GRU is described as:

$$\begin{aligned} h_{t-1} &= \text{OUT}_{t-1} \\ z_t &= \text{sigmoid}(W_z[h_{t-1}, X_t]) \\ r_t &= \text{sigmoid}(W_r[h_{t-1}, X_t]) \\ \tilde{h}_t &= \tanh(W[r_t h_{t-1}, X_t]) \\ h_t &= (1 - z_t)h_{t-1} + z_t \tilde{h}_t \\ \text{OUT}_t &= h_t \end{aligned}$$

11.31 References

- *Neural Computing: Theory and Practice* (1989). Philip D. Wasserman.
- MIT 6.034 (Fall 2010): Artificial Intelligence. Patrick H. Winston.
- <http://www.marekrei.com/blog/26-things-i-learned-in-the-deep-learning-summer-school/>

- Fundamentals of Deep Learning (2015). Nikhil Buduma.
- Understanding LSTM Networks. colah. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

12

Probabilistic Machine Learning

Fundamentally, machine learning is all about data:

- Stochastic, chaotic, and/or complex generative processes
- Noisily observed
- Partially observed

So there is a lot of uncertainty - we can use probability theory to express this uncertainty in the form of probabilistic models.

12.0.1 Probabilistic modeling

We have some data x_1, x_2, \dots, x_n and some latent variables y_1, y_2, \dots, y_n we want to uncover, which correspond to each of our data points.

We have a parameter θ .

A probabilistic model is just a parameterized joint distribution over all the variables:

$$P(x_1, \dots, x_n, y_1, \dots, y_n | \theta)$$

We usually interpret such models as a *generative* model - how was our observed data *generated* by the world?

So the problem of inference is about learning about our latent variables given the observed data, which we can get via the posterior distribution:

$$P(y_1, \dots, y_n | x_1, \dots, x_n, \theta) = \frac{P(x_1, \dots, x_n, y_1, \dots, y_n | \theta)}{P(x_1, \dots, x_n | \theta)}$$

Learning is typically posed as a *maximum likelihood* problem; that is, we try to find θ which maximizes the probability of our observed data:

$$\theta^{ML} = \operatorname{argmax}_{\theta} P(x_1, \dots, x_n | \theta)$$

Then, to make a prediction we want to compute the conditional distribution of some future data:

$$P(x_{n+1}, y_{n+1} | x_1, \dots, x_n, \theta)$$

Or, for classification, if we have some classes, each parameterizing a joint distribution, we want to pick the class which maximizes the probability of the observed data:

$$\operatorname{argmax}_c P(x_{n+1} | \theta^c)$$

In probabilistic learning, you typically have two separate problems:

1. Learning the parameters
2. Inference (computing the posterior distribution of the latent variables given the observed data and parameters)

12.0.2 Bayesian modeling

Bayesian modeling treats those two problems as one.

We first have a prior distribution over our parameters (i.e. what are the likely parameters?) $P(\theta)$.

From this we compute a posterior distribution which combines both inference and learning:

$$P(y_1, \dots, y_n, \theta | x_1, \dots, x_n) = \frac{P(x_1, \dots, x_n, y_1, \dots, y_n | \theta)P(\theta)}{P(x_1, \dots, x_n)}$$

Then prediction is to compute the conditional distribution of the new data point given our observed data, which is the marginal of the latent variables and the parameters:

$$P(x_{n+1} | x_1, \dots, x_n) = \int P(x_{n+1} | \theta)P(\theta | x_1, \dots, x_n)d\theta$$

Classification then is to predict the distributions of the new datapoint given data from other classes, then finding the class which maximizes it:

$$P(x_{n+1}|x_1^c, \dots, x_n^c) = \int P(x_{n+1}|\theta^c)P(\theta^c|x_1^c, \dots, x_n^c)d\theta^c$$

Bayesian probability modeling examples

These examples are all parametric models.

Model-based clustering

- model data from heterogeneous unknown sources
- K unknown sources (clusters)
- each cluster/source is modeled using a parametric model (e.g. a Gaussian distribution)

For a given data point i , we have:

$$z_i|\pi \sim \text{Discrete}(\pi)$$

Where z_i is the cluster label for which data point i belongs to. This is the latent variable we want to discover.

π is the *mixing proportions* which is the vector of probabilities for each class k , that is:

$$\pi = (\pi_1, \dots, \pi_K)|\alpha \sim \text{Dirichlet}\left(\frac{\alpha}{K}, \dots, \frac{\alpha}{K}\right)$$

That is, $\pi_k = P(z_i = k)$.

We also model each data point x_i as being drawn from a source (cluster) like so, where F is however we are modeling the cluster (e.g. a Gaussian), parameterized by $\theta_{z_i}^*$, that is some parameters for the z_i -labeled cluster:

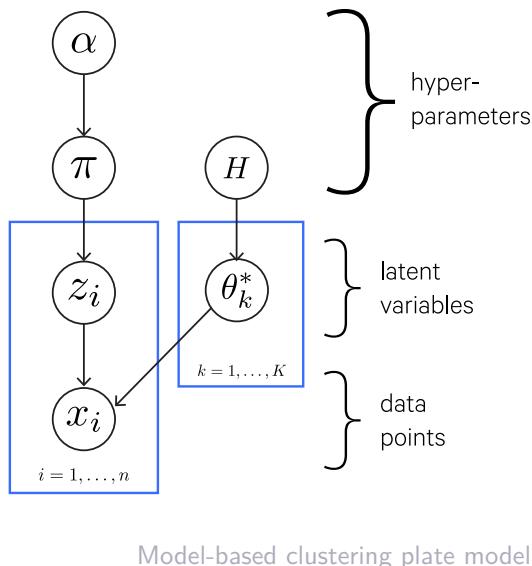
$$x_i|z_i, \theta_k^* \sim F(\theta_{z_i}^*)$$

(Note that the star, as in θ^* , is used to denote the optimal solution for θ .)

For this approach we have two priors over parameters of the model:

- For the mixing proportions, we typically use a Dirichlet prior (above) because it has the nice property of being a conjugate prior with multinomial distributions.
- For each cluster k we use some prior H , that is $\theta_k^*|H \sim H$.

Graphically, this is:



Model-based clustering plate model

Hidden Markov Models

HMMs can be thought of as clustering over time; that is, each state is a “cluster”.

The data points and latent variables are sequences, and π_k becomes the transition probability given the state (cluster) k . θ_k^* becomes the emission distribution for x given state k .

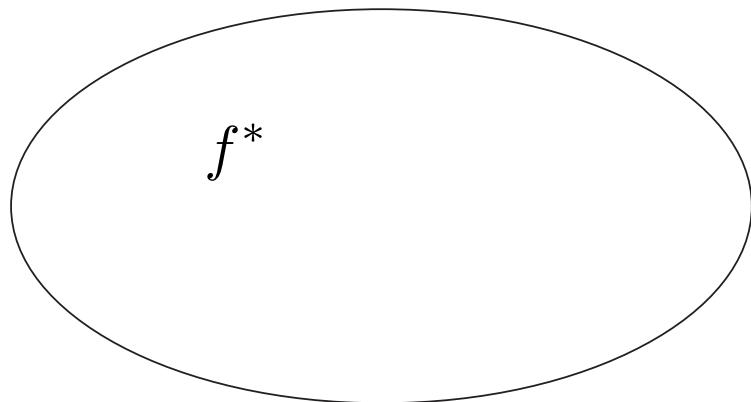
12.1 Nonparametric models

First: a parametric model is one in which the capacity is fixed and does not increase with the amount of training data. For example, a linear classifier, a neural network with fixed number of hidden units, etc.

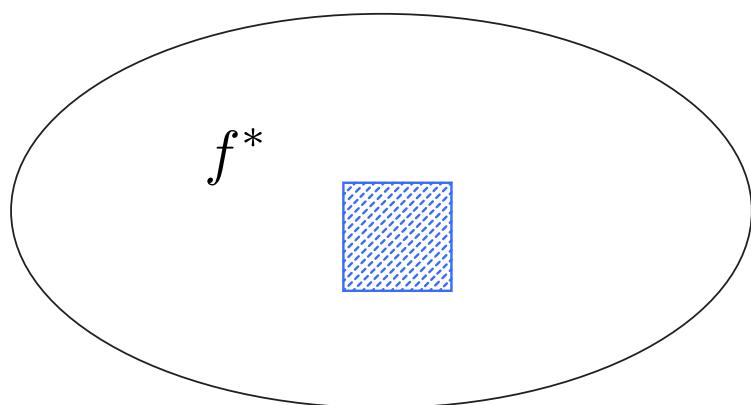
12.1.1 What is a nonparametric model?

- counterintuitively, it does not mean a model without parameters. Rather, it means a model with a very large number of parameters (e.g. infinite). Here, “nonparametric” refers more to “not a parametric model”, *not* “without parameters”.
- could also be defined as a parametric model where the number of parameters increases with the data, instead of fixing the number of parameters (that is, the number of things we can learn) as is the case with parametric models. I.e. the capacity of the model increases with the amount of training data.
- can also be defined as a family of distributions that is dense in some large space relevant to the problem at hand.
 - For example, with a regression problem, the space of possible solutions may be all continuous functions, which is infinite-dimensional (if you have infinite cardinality). A nonparametric model can span this infinite space.

To expand and visualize the last point, consider the regression problem example.



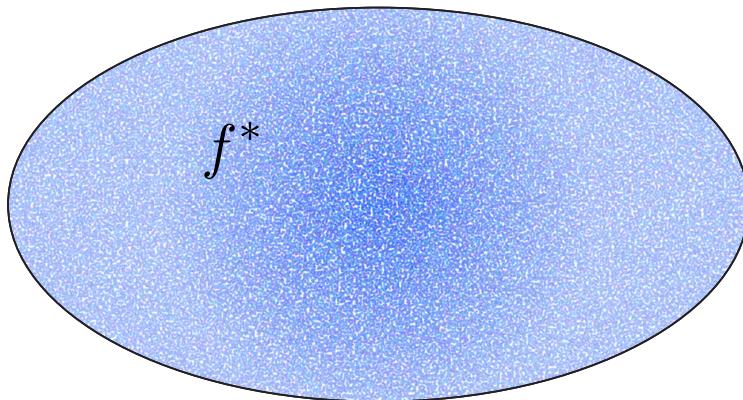
Space of continuous functions



Space of continuous functions w/ parametric model

This is the space of continuous functions, where f^* is the function we are looking for.

With a parametric model, we have a finite number of parameters, so we can only cover a fraction of this space (the orange square).



Space of continuous functions w/ nonparametric model

However, with a nonparametric model, we can have infinite parameters and cover the entire space. We apply some assumptions, e.g. favoring simpler functions over complex ones, so we can apply a prior to the space which assigns more mass to simpler functions (the darker parts in the accompanying figure). But every part of the space still has some mass.

12.1.2 Why use a Bayesian nonparametric approach?

1. Model selection

- e.g. clustering - you have to specify the number of clusters. Too many and you overfit, too few and you underfit.
- with a Bayesian approach you are not doing any optimizing (such as finding a maximum likelihood), you are just computing a posterior distribution. So there is no “fitting” happening, so you cannot overfit.
- If you have a large model or one which grows with the amount of data, you can avoid underfitting too.
- (of course, you can still specify an incorrect model and get poor performance)

2. Useful properties of Bayesian nonparametric models

- *Exchangeability* - you can permute your data without affecting learning (i.e. order of your data doesn't matter)
- Can model Zipf, Heap, and other power laws
- Flexible ways of building complex models from simpler parts

Nonparametric models still make modeling assumptions, they are just less constrained than most parametric models.

There are also *semiparametric* models in which they are nonparametric in some ways and parametric in others.

12.2 The Dirichlet Process

The Dirichlet process is “the cornerstone of Bayesian nonparametrics”.

It is a stochastic process - a model over an infinite collection of random variables.

There are a few ways to think about Dirichlet processes:

- the infinite limit of a Gibbs sampler for finite mixture models
- the Chinese Restaurant Process
- The stick-breaking construction

12.2.1 Finite Mixture Models

This is a continuation of the model-based Clustering approach mentioned earlier.

We want to learn, via inference, values for π , z_i , and θ_k^* .

We can use a form of MCMC sampling - Gibbs sampling.

(to do: this is incomplete)

12.2.2 Chinese Restaurant Process

Partitions

Given a set S , a partition ϱ is a disjoint family of non-empty subsets (clusters) of S whose union is S . So a partition is some configuration of clusters which encompasses the members of S .

E.g.

$$\begin{aligned} S &= \{A, B, C, D, E, F\} \\ \varrho &= \{\{A, D\}, \{B, C, E\}, \{F\}\} \end{aligned}$$

The set of all partitions of S is denoted \mathcal{P}_S .

Random partitions are random variables taking value in \mathcal{P}_S .

The Chinese Restaurant Process (CRP)

The CRP is an example of random partitions and involves a sequence of customers coming into a restaurant. Each customer decides whether or not to sit at a new (empty) table or join a table with other customers. The customers are sociable so prefer to join tables with more customers, but there is still some probability that they will sit at a new table:

$$P(\text{sit at new table}) = \frac{\alpha}{\alpha + \sum_{c \in \varrho} n_c}$$

$$P(\text{sit at table } c) = \frac{n_c}{\alpha + \sum_{c \in \varrho} n_c}$$

Where n_c is the number of customers at a table c and α is a parameter.

Here the customers correspond to members of the set S , and tables are the clusters in a partition ϱ of S .

This process has a *rich-get-richer* property, in that large clusters are more likely to attract more customers, thus growing larger, and so on.

If you multiply all the conditional probabilities together, the overall probability of the partition ϱ , called the *exchangeable partition probability function* (EPPF), is:

$$P(\varrho | \alpha) = \frac{\alpha^{|\varrho|} \Gamma(\alpha)}{\Gamma(n + \alpha)} \prod_{c \in \varrho} \Gamma(|c|)$$

This probability ends up not depending on the sequence in which customers arrive - so this is an *exchangeable random partition*.

The α parameter affects the number of clusters in the partition - the larger the α , the more clusters we expect to see.

Model-based Clustering with the Chinese Restaurant Process

Given a dataset S , we want to partition it into clusters of similar items.

Each cluster $c \in \varrho$ is described by a model $F(\theta_c^*)$, for example a Gaussian, parameterized by θ_c^* .

We model each item in each cluster as drawn from that cluster's model.

We are going to use a Bayesian approach, so we introduce a prior over ϱ and θ_c^* and the compute posteriors over both. We use a CRP mixture model; that is we use a Chinese Restaurant Process for the prior over ϱ and an independent and identically distributed (iid) prior H over the cluster parameters θ_c^* .

So the CRP mixture model in more detail:

- $\varrho \sim CRP(\alpha)$
- $\theta_c^* | \varrho \sim H$ for $c \in \varrho$
- $x_i | \theta_c^*, \varrho \sim F(\theta_c^*)$ for $c \in \varrho$ with $i \in c$

12.3 References

- Yee Whye Teh, MLSS 2013 (Max Planck Institute for Intelligent Systems, Tübingen) <https://www.youtube.com/watch?v=dNeW5zoNJ7g>

- IFT 725 Review of fundamentals
-

12.4 Infinite Mixture Models and the Dirichlet Process

(this is basically a paraphrasing of [this post by Edwin Chen](#).)

Many clustering methods require the specification of a fixed number of clusters. However, in real-world problems there may be an infinite number of possible clusters - in the case of food there may be Italian or Chinese or fast-food or vegetarian food and so on. Nonparametric Bayesian methods allow parameters to change with the data; e.g. as we get more data we can let the number of clusters grow.

Say we have some data, where each data point is some vector.

We can view our data from a generative perspective: we can assume that the true clusters in the data are each defined by some model with some parameters, such as Gaussians with μ_i and σ_i parameters. We further assume that these parameters themselves come from a distribution G_0 . Then we assume the data is generated by selecting a cluster, then taking a sample from that cluster.

Ok, how then do we assign the data points to groups?

12.4.1 Chinese Restaurant Process

(see explanation above)

(As a side note, the Indian Buffet Process is an extension of the CRP in which customers can sample food from multiple tables, that is, they can belong to multiple clusters.)

More formally:

- Generate table assignments $g_1, \dots, g_n \sim CRP(\alpha)$, that is, according to a Chinese Restaurant Process. g_i is the table assigned to datapoint i .
- We generate table parameters $\phi_1, \dots, \phi_m \sim G_0$ according to the base distribution G_0 , where ϕ_k is the parameter for the k th distinct group.
- Given the table assignments and table parameters, generate each datapoint $p_i \sim F(\phi_{g_i})$ from a distribution F with the specified table parameters. For example, F could be a Gaussian and p_i might be a vector specifying the mean and standard deviation.

12.4.2 Polya Urn Model

Basically the same as the Chinese Restaurant Process, except that while the CRP specifies a distribution over partitions (see above), the Polya Urn model does that and also assigns parameters to each group.

Say we have an urn containing $\alpha G_0(x)$ balls of some color x for each possible value of x . G_0 is our base distribution and $G_0(x)$ is the probability of sampling x from G_0 .

Then we iteratively pick a ball at random from the urn, place it back and also place an additional new ball of the same color of the one we drew.

As α increases (that is, we draw more new ball colors from the base distribution, which is the same as placing more weight on our prior), the colors in the urn tend towards the base distribution.

More formally:

- Generate colors $\phi_1, \dots, \phi_n \sim \text{Polya}(G_0, \alpha)$, that is, according to a Polya Urn Model. ϕ_i is the color of the i th ball.
- Given the ball colors, generate each datapoint $p_i \sim F(\phi_i)$ (where we are using F is a way like in the Chinese Restaurant Process above).

12.4.3 Stick-Breaking Process

The stick-breaking process is also very similar to the CRP and the Polya Urn model.

We start with a “stick” of length one, then generate a random variable $\beta_1 \sim \text{Beta}(1, \alpha)$. Since we’re drawing from the Beta distribution, β_1 will be a real number between 0 and 1 with the expected value $\frac{1}{1+\alpha}$.

Then break off the stick at β_1 . We define w_1 to be the length of the left stick.

Then we take the right piece (the one we broke off) and generate $\beta_1 \sim \text{Beta}(1, \alpha)$.

Then break off the stick at β_2 , set w_2 to be the length of the stick to the right, and so on.

Here α again functions as a dispersion parameter; when it is low there are few, denser clusters, when it is high, there are more clusters.

More formally:

- Generate group probabilities (stick lengths) $w_1, \dots, w_\infty \sim \text{Stick}(\alpha)$, that is, according to a Stick-Breaking process.
- Generate group parameters $\phi_1, \dots, \phi_\infty \sim G_0$, where ϕ_k is the parameter for the k th distinct group.
- Generate group assignments $g_1, \dots, g_n \sim \text{Multinomial}(w_1, \dots, w_\infty)$ for each datapoint.
- Given group assignments and group parameters, generate each datapoint $p_i \sim F(\phi_{g_i})$ (where we are using F is a way like in the Chinese Restaurant Process above).

12.4.4 Dirichlet Process

The CRP, Polya Urn Model, and Stick-Breaking Process are all connected to the Dirichlet Process.

Suppose we have a Dirichlet process $DP(G_0, \alpha)$ where G_0 is the base distribution and α is the dispersion parameter. Say we want to sample $x_i \sim G$, where G is a distribution sampled from our Dirichlet Process, $G \sim DP(G_0, \alpha)$.

We could generate these x_i values by taking a Polya Urn Model with color distribution G_0 and dispersion α - then x_i could be the color of the i th ball in the urn.

Or we could generate these x_i by assigning customers to tables via a CRP with dispersion α . Then all the customers for a table are given the same value (e.g. color) sampled from G_0 . x_i is the value/color given to the i th customer; here x_i can be thought of as the parameters for table i .

Or we could generate weights w_k via a Stick-Breaking Process with dispersion α . Then we give each weight w_k a value/color v_k sampled from G_0 . We assign x_i to v_k with probability w_k .

More formally:

- Generate a distribution $G \sim DP(G_0, \alpha)$ from a Dirichlet process with base distribution G_0 and a dispersion parameter α .
- Generate group-level parameters $x_i \sim G$ where x_i is the group parameter for the i th datapoint. Note that x_i is not the same as ϕ_i ; x_i is the parameter associated to the group that the i th data point belongs to whereas ϕ_k is the parameter of the k th distinct group.
- Given group-level parameters x_i , generate each datapoint $p_i \sim F(x_i)$ (where we are using F is a way like in the Chinese Restaurant Process above).

12.4.5 Gibbs Sampling

Now that we have the generative model, we can use it to calculate the probability of some set of group assignments for our data points. But how do we learn what a good set of group assignments is?

We can use Gibbs Sampling, that is:

- Take the set of data points, and randomly initialize group assignments.
- Pick a point. Fix the group assignments of all the other points, and assign the chosen point a new group (which can be either an existing cluster or a new cluster) with a CRP-ish probability (as described in the models above) that depends on the group assignments and values of all the other points.
- We will eventually converge on a good set of group assignments, so repeat the previous step until happy.

12.4.6 References

- Edwin Chen, <http://blog.echen.me/2012/03/20/infinite-mixture-models-with-nonparametric-bayes-and-t>

12.5 Parametric models vs nonparametric models

Parametric models are relatively rigid; once you choose a model, there are some limitations to what forms that model can take (i.e. how it can fit to the data), and the only real flexibility is in the

parameters which can be adjusted. For instance, with linear regression, the model must take the form of $y = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$; you can only adjust the β_i values. If the “true” model does not take this form, we probably won’t be able to estimate it well because the model we chose fundamentally does not conform to it.

Parametric models, on the other hand, offer greater freedom of fit.

As an example, a histogram can be considered a nonparametric representation of a probability density - it “lets the data speak for itself”, so to speak (you may hear nonparametric models described in this way). The density that forms in the histogram is determined directly by the data. You don’t make any assumptions about what the distribution is beforehand - e.g. you don’t have to say, “I think this might be a normal distribution”, and then try to force the normal probability density function onto the data.

Nonparametric models don’t actually mean there are no parameters, but it is perhaps better described as not having a fixed set of parameters.

12.5.1 References

- Kernel Density Estimation and Kernel Regression. Justin Esarey. <https://www.youtube.com/watch?v=QSNN0no4dSI>

Part III

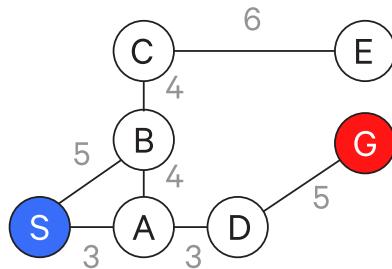
Artificial Intelligence

13

Search

13.1 Depth-First, Hill Climbing, Beam

Consider the following search space, where S is our starting point and G is our goal:



Example search space

You can illustrate paths in your search problem using a tree.

13.1.1 “British Museum” search

Exhaustively search all paths (without revisiting any previously visited points).

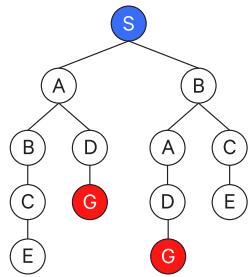
13.1.2 Depth-First Search

Go down the left branch of the tree (by convention) until you can't go any further.

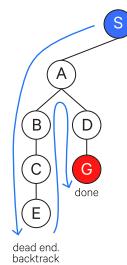
If that is not your target, then *backtrack* - go up to the closest branching node and take the other leftmost path.

Repeat until you reach your target.

It stops just on the first complete path, which may not be the optimal path.



“British Museum” search



Depth-first search

Another way to think about depth-first search is with a *queue* which holds your candidate paths as you construct them.

Your starting “path” includes just the starting point:

$[(S)]$

Then on each iteration, you take the left-most path (which is always the first in the queue) and check if it reaches your goal.

If it does not, you extend it to build new paths, and replace it with those new paths.

$[(SA), (SB)]$

On this next iteration, you again take the left-most path. It still does not reach your goal, so you extend it. And so on:

$[(SABC), (SAD), (SB)]$
 $[(SABCE), (SAD), (SB)]$

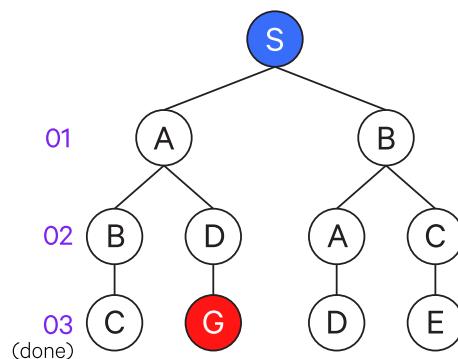
You can no longer extend the left-most path, so just remove it from the queue.

$[(SAD), (SB)]$

Then keep going.

13.1.3 Breadth-First Search

Build out the tree level-by-level until you reach your target.



Breadth-first search

In the queue representation, the only thing that is different from depth-first is that instead of placing new paths at the front of the queue, you place them at the back.

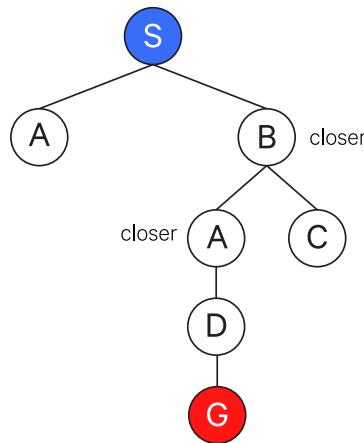
Breadth-first with extended list filtering

We can modify the queue algorithm to only extend the first path on the queue if its final node has not yet been extended. That is, if we are extending a node, and its final node has already been extended in some other path, we ignore the rest of that path - it would be redundant to re-build a path from that node. This is called using an *extended list*.

You can apply this same filtering technique to depth-first search as well.

13.1.4 Hill-Climbing Search

Hill-climbing is similar to depth-first search, but rather than just sticking to the left-most path, at any branching point you select the node which is closest to your goal.



Hill-climbing search

In this case of the example this isn't the optimal path (that is, hill-climbing can get stuck in local maxima), but it is possible for hill-climbing to find the optimal path. There isn't any backtracking in hill-climbing.

The queue representation is the same as depth-first search, but you sort your queue by distance to the goal.

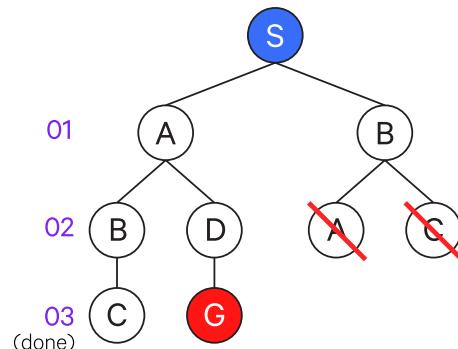
Note that *distance* here is the *heuristic distance*, also called *airline distance*, which is the straight and direct distance between two points (“as the crow flies”).

13.1.5 Beam Search

Beam search is similar to breadth-first search. You set a *beam width* w which is the limit to the number of paths you will consider at any level. This is typically a low number like 2.

Then you proceed like a beam search and look at which w nodes in the new level get you closest (in terms of heuristic distance) to your goal.

The queue representation is the same as breadth-first search, but you only keep the w best paths.

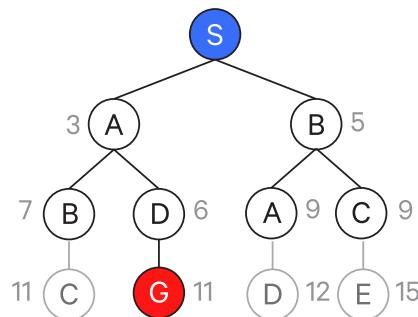


Beam search

13.2 Optimal, Branch, and Bound

13.2.1 Branch & Bound Search and the A* algorithm

On each iteration, extend the shortest cumulative path. Once you reach your goal, extend every other extendible path to check that its length ends up being longer than your current path to the goal.



Branch and bound search

In the queue representation, you would test the first path to see if it reaches the goal, if not, then extend the first path and sort by its path length, shortest at the front.

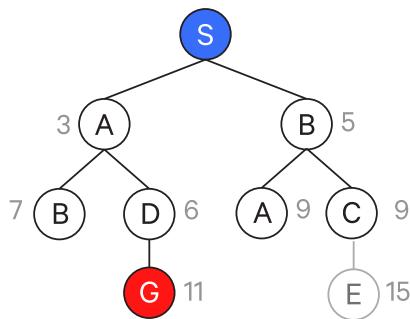
This approach can be quite exhaustive, but it can be improved.

We can augment branch & bound by using an extended list, in which we extend the first path and sort by path length only if its last node has not already been extended elsewhere. This prevents redundant traversing of already-traversed paths.

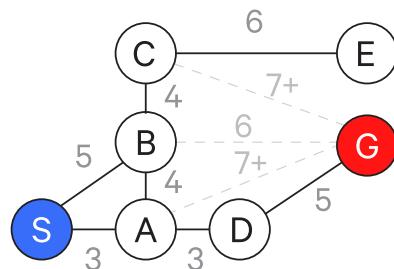
We can use airline distances in combination with the cumulative distance thus far in order to calculate the lower bound for the complete path (if one exists). That is because the airline distance is the straight line from the current node to the goal; that is, it is the shortest possible distance to the goal. However, it is not certain if there exists a path from the current node to the goal. This technique is called the *admissible heuristic*.

When we combine the extended list and the admissible heuristic with branch and bound, we get the *A* algorithm*.

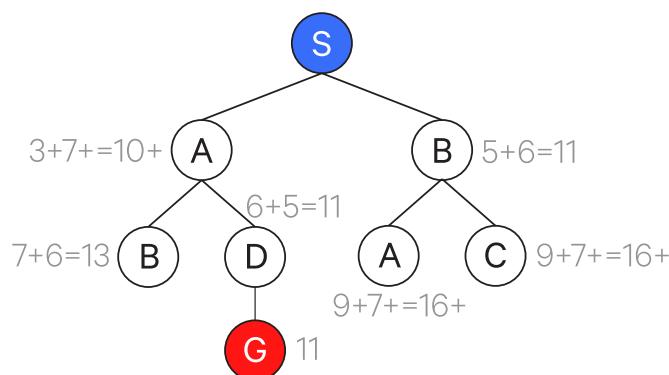
The algorithm is slightly different:



Branch and bound search w/ extended list



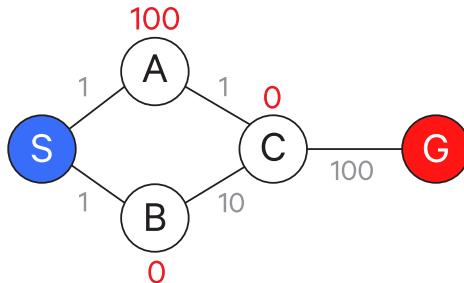
Search space with airline distances noted. 7+ means slightly more than 7.



Branch and bound w/ the admissible heuristic

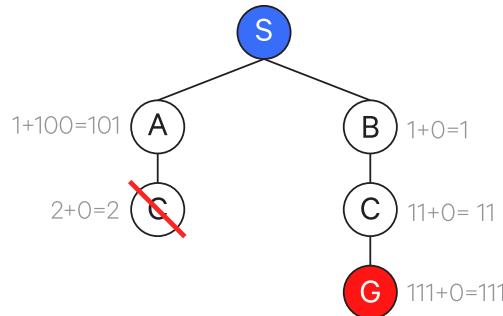
- Initialize the queue
- Test the shortest path (rather than the first path, as we did previously)
- Extend the shortest path

Thus far we have been working with maps. But search is not restricted to maps. The admissible heuristic falls apart in certain problems. Consider the following non-euclidean graph:



Non-euclidean search

The airline distances from a node to the goal are marked in red.



Admissible heuristic in non-euclidean search

Here if we use the admissible heuristic, we end up taking the path $SBCG$. When we do our checking phase to try the path starting with SA , we end up at C and stop there because we have already checked C . Thus we miss out on the optimal path.

Formally, we can define the admissible heuristic as:

$$H(x, G) \leq D(x, G)$$

That is, a node is admissible if the estimated distance $H(x, G)$ between node x and the goal G is less than or equal to the actual distance $D(x, G)$ between the node and the goal.

We can set a stronger condition for the previous non-euclidean example. This condition is called *consistency*:

$$|H(x, G) - H(y, G)| \leq D(x, y)$$

That is, the absolute value of the difference between the estimated distance between a node x and the goal and the estimated distance between a node y and the goal is less than or equal to the distance between the nodes x and y .

13.3 Games, Minimax, and Alpha-Beta

Games (like chess) are useful for modeling some aspects of intelligence.

There are a few ways you could design a chess-playing program.

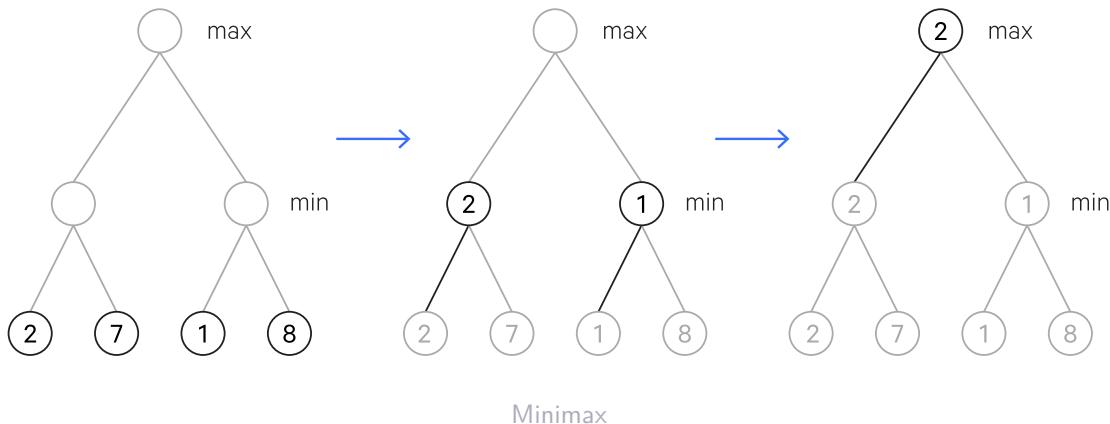
You could use if-then rules but these are limited for games as complex as chess.

You could take the *look ahead* approach look at the next possible board states. You'd take various features of the chessboard, f_1, f_2, \dots, f_n and pass them into a *static function*, $s = g(f_1, f_2, \dots, f_n)$, typically a linear polynomial (e.g. $c_1 f_1 + c_2 f_2 + \dots + c_n f_n$), so called "static" because it only considers a static state of the board. It is also called a *linear scoring polynomial* because it is used to assign some value to some board state. Then at a given step you can pick the move which leads to the highest-scoring board state.

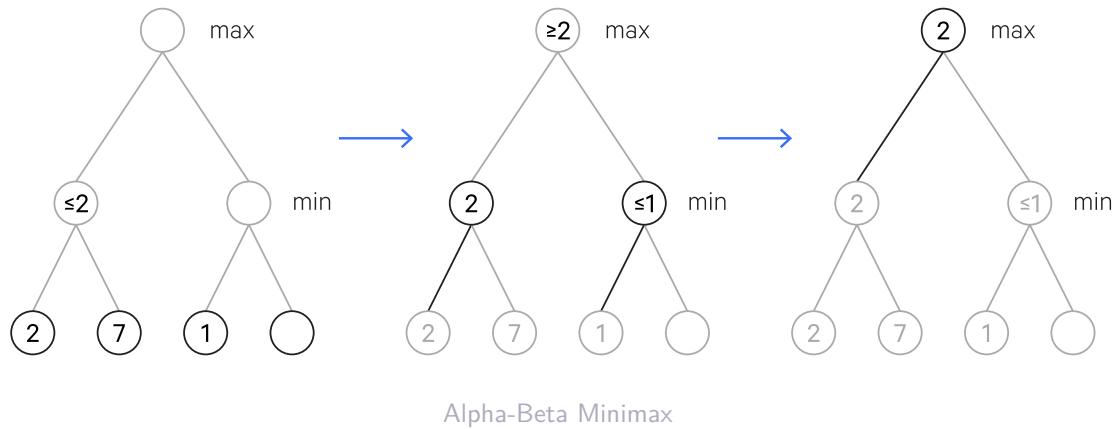
You could evaluate a tree of all possible outcomes from the current state. The number of possibilities considered at each level is called the *branching factor* b , the number of levels considered is the *depth* d . The total number of terminal nodes (possible outcomes) is just b^d . For chess there typically is a branching factor of 10 and a depth of 120 to consider, which is a very large number (10^{120}), far too large to compute.

You could instead just do look ahead but go a few steps further, computing the scores for the leaf nodes using a static function, then selecting a branch path using the *minimax* approach.

In the minimax approach you start at the bottom of the tree (where you have your scored leaf nodes). You compute the scores of the (branch) nodes the next level up by selecting the *minimum* of each of their children. Then you compute the scores for the nodes the next level up by selecting the *maximum* of each of their children. You continue alternating in this way up the tree.



Ideally there would be a way to apply minimax without having to exhaustively consider all branches or compute all static values at the leaf nodes. And there is - it is the *alpha-beta* algorithm, which is an augmentation of minimax.



Here we can look at branching and figure out a bound for describing its score.

First we look at the left-most branch and see the value 2 in its left-most leaf node. Since we are looking for the min here, we know that the score for this branch node will be at most 2. If we then look at the other leaf node, we see that it is 7 and we know the branch node's score is 2.

At this point we can apply a similar logic to the next node up (where we are looking for the max). We know that it will be at least 2.

So then we look at the next branch node and see that it will be at most 1. We don't have to look at the very last leaf node because now we know that the max node can only be 2. So we have saved ourselves a little trouble.

In larger trees this approach becomes very valuable, since you are effectively discounting entire branches and saving a lot of unnecessary computation. This allows you to compute deeper trees.

You may want an “insurance policy” for time-sensitive minimax searches. Say you want to compute down to depth d but don’t have enough time - well you could just fall back to the result calculated from the depth $d - 1$ (or whatever depth that has been reached by the time an answer is needed). This is called *progressive deepening* (or *iterative deepening*). This type of algorithm is called an *anytime algorithm* because it has an answer ready at anytime.

In these examples, the trees have been developed *evenly*, that is, each branch goes down to the same depth. But it doesn’t have to be that way - there may be situations in games where you want to expand a certain branch to a further depth - this is called *uneven tree development* and was an additional feature of Deep Blue.

As effective this strategy is at playing chess, this is not how human chess players play chess - they develop pattern recognition repertoires in which they can recognize game board states and decide how to move based on that.

13.4 Constraints: Search, Domain Reduction

Constraint propagation is a method in which constraints eliminate possible choices, thus reducing a search space. For example, the possible values for x are 1, 2, 3, 4 but I know the constraint $x > 2$. Thus I have reduced my search space to 3, 4.

Some vocabulary:

- a variable v is something that can have assignment
- a value x is something that can be assigned (e.g. is a label)
- a domain D is a bag of values
- a constraint C is a limit on variable values

The general algorithm is as follows:

- For each depth-first search assignment
 - For each variable v_i considered*
 - * For each x_i in D_i
 - For each constraint $C(x_i, x_j)$ where $x_j \in D_j$
 - If $\nexists x_j \mid C(x_i, x_j)$ is satisfied, remove x_i from D_i
 - If D_i empty, backup

* There are a number of ways to define which variables are considered:

1. consider nothing (which defeats the point of the constraints, since you aren't checking that they are satisfied)
2. consider only the assigned variable
3. consider neighbors
4. propagate checking through variables whose domains have shrunk to a single value (that is, check neighbors if the variable's domain has been reduced to a single value, and repeat as necessary, that is, if those neighbors' domains have all been reduced to a single value, keep going, etc) (works well)
5. propagate checking through variables whose domains have been reduced by any amount
6. consider all variables (too much work)

You can also change the order in which you consider variables - if you consider the most constrained variables first, ordinary depth-first search will work, but you might as well combine these approaches.

13.5 Monte Carlo Tree Search

An alternative to minimax is **Monte Carlo Tree Search**, which does not require exhaustively considering all branches.

13.5.1 Multi-armed bandit

Say you have multiple options with uncertain payouts. You want to maximize your overall payout, and it seems the most prudent strategy would be to identify the one option which consistently yields better payouts than the other options.

However - how do you identify the best option, and do so quickly?

This problem is known as the *multi-armed bandit* problem, and a common strategy is based on upper confidence bounds (UCB).

To start, you randomly try the options and compute confidence intervals for each options' payout:

$$\bar{x}_i \pm \sqrt{\frac{2 \ln(n)}{n_i}}$$

where:

- \bar{x}_i is the mean payout for option i
- n_i is the number of times option i was chosen
- n is the total number of trials

You take the upper bound of these confidence intervals and continue to choose the option with the highest upper bound. As you use this option more, its confidence interval will narrow (since you have collected more data on it), and eventually another option's confidence interval upper bound will be higher, at which point you switch to that option.

13.5.2 Monte Carlo

Say you are at some arbitrary position in your search tree (it could be the start or somewhere further along). You can treat the problem of what node to move to next as a multi-armed bandit problem.

At first, you have no statistical information about the child nodes to compute confidence intervals. So you randomly choose a child and run Monte Carlo simulations down that branch to see the outcomes.

For each simulation run, you go along each node in the branch that was walked and increment its play count (i.e. number of trials) by 1, and if the outcome is a win, you increment its win count by 1 as well

You repeat this until you have enough statistics for the direct child nodes of your current position to make a UCB choice as to where to move next.

You will need to run less simulations over time because you accumulate these statistics for the search tree.

13.6 References

- MIT 6.034 (Fall 2010): Artificial Intelligence. Patrick H. Winston.
- http://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/?utm_campaign=Data%2BElixir&utm_medium=email&utm_source=Data_Elixir_53 Planning is tricky be-

cause:

- environmental properties:
 - systems may be *stochastic*
 - there may be *multiple agents* in the system
 - there may be *partial observability* (that is, the state of the system may not be fully known)
- agent properties:
 - some information may be *unknown*
 - plans are hierarchical (high level to low level parts)

In planning we represent the world in *belief states*, in which multiple states are possible (because of incomplete information, we are not certain what the true state of the world is). Actions that are taken can either increase or decrease the possible states, in some cases down to one state.

Sequences of actions can be defined as trees, where each branch is an action and each node is a state or is an observation of the world. Then we search this tree for a plan which will satisfy the goal.

13.7 References

- Logical Foundations of Artificial Intelligence (1987) (Chapter 12: Planning)
-

13.8 References

- Planning Algorithms. Steven M. LaValle, 2006. <http://planning.cs.uiuc.edu/booka4.pdf>

Part IV

Simulation

Part V

Practice

14

Process

1. Data is never clean
2. You will spend most of your time cleaning and preparing data.
3. 95% of the tasks do not require deep learning.
4. 90% of cases GLM will do the trick.
5. Big Data is just a tool.
6. You should embrace the Bayesian approach.
7. No one cares how you did it.
8. Academia and business are two different worlds.
9. Presentation is key - be a master of PowerPoint.
10. All models are wrong, but some are useful.
11. There is no full automated data science. You need to get your hands dirty.

[source](#)

14.1 Data analysis approach

- Clearly and unambiguously define the question you wish to solve
- Determine the ideal dataset for your goal
 - **Descriptive** - whole population
 - **Exploratory** - random sample with multiple variables measured
 - **Inferential** - drawing a conclusion about a larger population from a random sample
 - **Predictive** - need training and testing data from the same population to build a model and a classifier
 - **Causal** - experimental data from a randomized study
 - **Mechanistic** - data from all components of the system you want to describe

- Obtain the data that you need - you may need to pay for it, or generate it yourself
- Clean the data and convert it to a format suitable for your needs
- Exploratory data analysis - become familiar with the data
- Statistical modeling/prediction
- Interpret results, with detailed explanations
- Challenge your results (and entire process), audit the whole thing. Try to come up with alternative analyses, and so on.
- Synthesize and write up your results
- Create reproducible code

From [Reproducible Research Course Notes](#), Xing Su.

15

Data Collection

When you have a research question, it is probably about a population as a whole. However, you are only ever able to collect a sample. You want to collect your sample in such a way that the population estimates you compute from the sample are as accurate as possible.

15.1 Sampling

To do so, you want to collect your sample randomly. However, even then there is possibility of introducing some bias into the sample, which can cause your sample to be *non-representative*.

A couple examples of bias:

- **non-response bias** - if collect data via surveys, and non-response is high, the collected results might not really reflect the population. Perhaps there are common factors which led many to not respond; perhaps these factors will influence your study in an unintentional way.
- **convenience bias** - similarly, your sample may only reflect cases which were more accessible due to your collection methodology.

There are a few different ways of randomly collecting your sample:

- **simple random** sampling - just randomly select your sample
- **stratified** sampling - divide the population into **strata**, which are groups of similar cases. Then simple random sampling is used within each stratum. All strata must be sampled from. One complication is that stratified samples are analyzed differently than simple random samples.
- **cluster** sampling - divide the population into clusters, randomly sample a fixed number of clusters, then collect simple random samples from each cluster. Not all clusters must be sampled from. Again, however, cluster samples require different analysis than simple random samples.

15.2 Studies

Broadly, data may be collected in one of two ways, which are further broken down into categories:

- **observational** study - the data collection process does not interfere with the population being studied
 - **prospective** study - identify individuals and collect data as they happen, going forward
 - **retrospective** study - use data which has already been collected, such as archived data
- **experimental** study - the data collection process involves some kind of intervention, the effect of which is under study

The only data you have available to analyze is the data that you collect. That is, you can only examine the variables for which you have collected data. Perhaps you want to see how two of them are correlated. But there is always the possibility of a **confounding variable**; that is, some variable that correlates with both. Say you see that *A* is correlated with *B* and you suspect there may be a causal relationship. Well there may be a third variable *C* which correlates with both *A*, *B* and might be the underlying cause.

15.3 References

- OpenIntro Statistics, Second Edition. David M Diez, Christopher D Barr, Mine Çetinkaya-Rundel.

o

15.4 Learning: Tips

(my main takeaways from the Sparse Spaces, Phonology lecture)

An ideal approach to AI problems is:

1. Specify the problem
2. Devise a representation suited to the problem
3. Determine an approach or method
4. Pick a mechanism or devise an algorithm
5. Experiment

This isn't a linear process; throughout working on the problem you are likely to jump around, perhaps redefine the problem, etc. But generally, this is sequence you'd want to go in.

However, in practice many AI practitioners fall in love with a particular mechanism or algorithm and try to apply it to everything, which is too inflexible approach - you should match the mechanism to the problem, not vice versa.

Coming up with the right representation is crucial to success in these kinds of problems. So how do you do it? There are a few heuristics about what makes a good representation:

- It makes the right things (distinctive features, relationships, etc) explicit
- It exposes constraints to work off of
- There is a localness - more compact rather than spread out

15.4.1 References

- MIT 6.034 (Fall 2010): Artificial Intelligence. Patrick H. Winston.

16

Data Visualization

16.1 Bivariate charts

Bivariate charts display information about the relationship between *two* variables. This includes scatterplots and line graphs.

16.2 Histograms

When you have continuous data, you can get a sense of the data's density around values using a histogram. A histogram breaks your data up into **bins**, so that values falling within each bin are grouped together. Then the number of values (i.e. the frequency of values) in each bin is plotted out like a bar chart. This helps provide a sense of how the data is distributed.

16.3 Scatterplots

Scatterplots plot out each individual data point (according to two dimensions) and can provide intuitions into correlation of the plotted dimensions, amongst other things.

16.4 References

- OpenIntro Statistics, Second Edition. David M Diez, Christopher D Barr, Mine Çetinkaya-Rundel. ## Poisson distribution

```
from scipy import stats
lmbda = 3 # mean
y = stats.poisson.pmf(range(10), lmbda)
plt.plot(y, 'ro')
```

16.5 Normal distribution

```
import numpy as np
from scipy import stats
mu, sig = 10, 3
xs = np.random.linspace(-10, 30)
y = stats.norm.pdf(xs, mu, sig)
plt.plot(xs, y, 'ro')
```

Part VI

Appendices

17

Data analysis with pandas

Basic concepts:

- a table with multiple columns is a DataFrame
- a single column on its own is a Series

Basic pandas commands for analyzing data. Good for use in iPython notebooks.

Note: columns here are ambiguous in their datatypes; these are just illustrations.

```
import pandas as pd
import pylab as plt

# Load data from csv into a DataFrame
df = pd.read_csv('data.csv')

# Get top 5 rows
df.head()

# Get top 10 rows
df.head(10)

# Get bottom 5 rows
df.tail()

# Get bottom 10 rows
df.tail(10)

# See overview of data
# (columns, number non-null values, datatypes)
```

```

df.info()

# See number of rows
len(df)

# See columns
df.columns

# Select a column (as a Series)
df['col_1']

# Operate on rows of the column
df['col_1'] + 10
df['col_1'] * 10
df['col_1'] + df['col_2']
# etc

# Round values of a column
df['col_1'].round()

# Create a new column
df['col_3'] = df['col_1'] + 10

# Sort by a column
df.sort('col_1')

# Set an index (the column you use becomes the index)
# This overwrites the existing index
df.set_index('col_1')

# Set an index w/o overwriting the existing index (append)
df.set_index('col_1', append=True)

# Set multiple indices
df.set_index(['col_1', 'col_2'])

# For speed benefits, you should sort your index
df.sort_index()

# Access item(s) by index value
df.loc['some value']

# Remove an index/indices
df.reset_index('col_1')

```

```
df.reset_index(['col_1', 'col_2'])

# Select multiple columns
df[['col_1', 'col_2']]

# See datatypes
df.dtypes

# See some descriptive statistics
# (e.g. count, mean, std, etc)
df.describe()

# Descriptive statistics for a single column
df['col_1'].describe()

# Get a particular descriptive statistics
df['col_1'].mean()
df['col_1'].median()
df['col_1'].mode()
df['col_1'].std()
df['col_1'].max()
df['col_1'].min()

# See unique values for a column
df['col_1'].unique()

# Count num of occurences for values in a column
df['col_1'].value_counts()

# Generate cross tab of two columns
pd.crosstab(df['col_1'], df['col_2'])

# Convert a string representation to a number representation
str_vals = df['col_1'].unique()
mapping = dict(zip(str_vals, range(0, len(str_vals) + 1)))
df['col_1_as_int'] = df['col_1'].map(mapping).astype(int)

# Select rows where a value satisfies a condition
df[df['col_1'] == True]

# Select rows where values satisfy multiple conditions
df[(df['col_1'] == True) & (df['col_2'] == True)] # and
df[(df['col_1'] == True) | (df['col_2'] == True)] # or
```

```

# Count rows
df[df['col_1'] == True].count()

# Get rows where columns have non-null values
df[df['col_1'].notnull()]

# Get rows where a column has a null value
df[df['col_1'].isnull()]

# Drop rows with any null values
df.dropna(axis=0, how='any')

# Drop rows of all null values
df.dropna(axis=0, how='all')

# Drop columns with null values
df.dropna(axis=1)

# Replace column's nan values with average for the column (in place)
df.replace({
    'col_1': {nan: df['col_1'].mean()}
}, inplace=True)

# Or more easily, replace nan values
df['col_1'].fillna(df['col_1'].mean(), inplace=True)

# Use "forward fill" (take previous non-nan values
# and fill downward until next non-nan value)
df['col_1'].fillna(method='ffill')

# Use "backward fill"
df['col_1'].fillna(method='bfill')

# Combine two dataframes (i.e. add columns)
pd.concat([df, df2], axis=1)

# Stack two dataframes (i.e. add rows)
pd.concat([df, df2], axis=0)

# Group by column
df.groupby('col_1')

# Group by columns
df.groupby(['col_1', 'col_2'])

```

```
# Count number of rows in groups
# (can also do all the other descriptive statistics)
df.groupby('col_1').size()

# For example, get max of each group:
df.groupby('col_1').max()

# Get mean of each group:
df.groupby('col_1').mean()
# etc

# Use multiple descriptors
df.groupby('col_1').agg(['min', 'max'])

# You can also group by manipulations of columns
# For example, group by decade
df.groupby(df['year'] // 10 * 10)

# Drop a column
df.drop('col_1', axis=1)

# Drop multiple columns
df.drop(['col_1', 'col_2'], axis=1)

# Get data as numpy array
df.values

# Find rows that satisfy a string condition
df['col_1'].str.startswith('foo')
df['col_1'].str.endswith('foo')
df['col_1'].str.contains('foo')
df['col_1'].str.len() > 10
df['col_1'].str.slice(0, 5)
# etc

# Turn rows or indices into columns
# (useful when doing groupbys with multiple columns)
df.unstack('col_1')

# Turn columns into indices
df.stack()

# Merge dataframes (automatically looks for matching columns)
```

```
df.merge(another_df)

# You can be explicit about matching columns
df.merge(another_df, on=['col_1', 'col_2'])
```

17.1 Dealing with datetimes

```
# Turn years into decades
df['year'] // 10 * 10

# Datetime slice (requires that your DataFrame has a DateTimeIndex)
start = datetime(year=1980, month=1, day=1, hour=0, minute=0)
end = datetime(year=1981, month=1, day=1, hour=0, minute=0)
df[start:end]

# Group by some time interval,
# assuming DateTimeIndex
df.groupby(pd.TimeGrouper(freq='M')) # month
df.groupby(pd.TimeGrouper(freq='10Min')) # 10 minutes
# etc

# Access different datetime resolutions
df['date_col'].dt.year
df['date_col'].dt.month
df['date_col'].dt.day
df['date_col'].dt.dayofyear
df['date_col'].dt.dayofweek
df['date_col'].dt.weekday
# etc

# For DateTime indices, you can do something similar
df.index.weekday
# etc

# Combine rows by some interval
df.resample('M', how='mean') # month
df.resample('M', how=np.median) # month
# etc
```

17.2 Loading data

```
# Load from CSV
```

```

df = pd.read_csv('data.csv')

# If you have an index column in the file, you can do:
df = pd.read_csv('data.csv', index_col='col1')

# Directly parse dates
df = pd.read_csv('data.csv', parse_dates=True)
df = pd.read_csv('data.csv', parse_dates=['date'])

# Skip rows
df = pd.read_csv('data.csv', skiprows=10)

```

17.3 Plotting

17.3.1 Initial setup

```

import matplotlib.pyplot as plt

# Set the global default size of matplotlib figures
plt.rc('figure', figsize=(10, 5))

```

17.3.2 Basics

```

# Plot a Series (x=index, y=values)
sr.plot()

# Plot a Dataframe
df.plot(x='col_1', y='col_2')

# Specify figsize
df.plot(figsize=(15,10))

# Specify y limits (similar for x limits)
df.plot(ylim=[0, 100])

# Plot multiple columns against the index
df.plot(subplots=True)

```

17.3.3 Plot a cross tab

```

# Generate cross tab of two columns
xt = pd.crosstab(df['col_1'], df['col_2'])

```

```
# Normalize
xt_norm = xt.div(xt.sum(1).astype(float), axis=0)

# Plot as stacked bar chart
xt_norm.plot(kind='bar',
stacked=True,
title='cross tab plot')
plt.xlabel('col_1')
plt.ylabel('col_2')
```

17.3.4 Plot subplots as a grid

```
# Plot a grid of subplots
fig = plt.figure(figsize=(10, 10))
fig_dims = (3,2)

plt.subplot2grid(fig_dims, (0, 0))
df['col_1'].value_counts().plot(kind='bar', title='column 1')

plt.subplot2grid(fig_dims, (0, 1))
df['col_2'].hist()
plt.title('histogram of column 2')
```

17.3.5 Plot overlays

```
for i in range(3):
    df[df['col_1'] == i].plot(kind='kde', alpha=0.5)
plt.title('overlaid density plots')
plt.legend(('class 1', 'class 2', 'class 3'), loc='best')
```

17.3.6 Other plots

```
# Scatter plot
plt.scatter(df['col_1'], df['col_2'])
plt.title('scatter plot')
plt.xlabel('column 1')
plt.ylabel('column 2')

# Histogram
bin_size = 10
max_val = df['col_1'].max()
df['col_1'].hist(bins=max_val/bin_size, range=(1, max_val))
```

```
# Stacked histogram
df1 = df[df['col_1'] == True]['col_2']
df2 = df[df['col_1'] == False]['col_2']
plt.hist([df1, df2], bins=max_val/bin_size, range=(1, max_val))
plt.legend(['true col_2', 'false col_2'], loc='best')
```

17.3.7 Decorating plots

```
plt.title('some title')
plt.xlabel('column 1')
plt.ylabel('column 2')
plt.grid(True)
plt.legend(['col 1', 'col 2'], loc='best')
```

17.3.8 Saving a figure

```
plt.savefig('filename.png')
```

17.4 iPython Notebooks

Run the iPython notebook server:

```
$ ipython notebook --pylab inline
```

The `--pylab inline` option renders plots in the notebook.

Alternatively, you can add:

```
%matplotlib inline
```

To the first cell in the notebook.

iPython notebooks have a command and edit (insert) mode.

Hit `esc` to enter command mode, and hit `h` to see the available commands.

Some useful commands:

- `A`: insert cell above
- `B`: insert cell below
- `shift+enter`: execute current cell and go to next cell (also works in edit mode)
- `ctrl+enter`: execute current cell (also works in edit mode)