

# **PROGRAMMING INTEGRATION PROJECT**

## **(CO3101) - Detect AI Generated Text**

by

Nguyen Ngoc Khoi, Nguyen Minh Khoi, Nguyen Quang Phu

Submitted to the Department of Computer Science  
in partial fulfillment of the requirements for the degree of  
**BACHELOR OF SCIENCE IN COMPUTER SCIENCE**

at the

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY - VNU-HCM

September 2024

© 2024 Nguyen Ngoc Khoi, Nguyen Minh Khoi, Nguyen Quang Phu. All rights reserved.

Authored by: Nguyen Ngoc Khoi, Nguyen Minh Khoi, Nguyen Quang Phu  
Department of Computer Science  
December, 2024

Certified by: Nguyen An Khuong  
Doctor of Philosophy in Mathematics, Thesis Supervisor

Accepted by: Department of Computer Science  
Faculty of Computer Science and Engineering  
Ho Chi Minh City University of Technology - VNU-HCM



# PROGRAMMING INTEGRATION PROJECT

SUPERVISOR

**Nguyen An Khuong**  
*University of Technology (HCMUT) VNU-HCM*  
*Faculty of Computer Science and Engineering*  
*PhD*

STUDENT

**Nguyen Ngoc Khoi**  
*2252378*  
*Faculty of Computer Science and Engineering*

**Nguyen Minh Khoi**  
*2252377*  
*Faculty of Computer Science and Engineering*

**Nguyen Quang Phu**  
*2252621*  
*Faculty of Computer Science and Engineering*



# Contents

Title page	1
List of Figures	25
List of Tables	31
1 Abstract	33
2 Description	35
3 Basic Math and Linear Algebra	37
3.1 Notation . . . . .	37
3.1.1 Set notation . . . . .	37
3.1.2 Miscellaneous symbols . . . . .	38
3.1.3 Operations . . . . .	38
3.1.4 Functions . . . . .	38
3.1.5 Exponential and logarithmic functions . . . . .	39
3.1.6 Circular and hyperbolic functions . . . . .	39
3.1.7 Complex numbers . . . . .	39
3.1.8 Matrices . . . . .	40
3.1.9 Vectors . . . . .	40
3.2 Convex Sets . . . . .	41
3.2.1 Definition . . . . .	41
3.2.2 Examples . . . . .	42
3.2.3 Intersection of Convex Sets . . . . .	43
3.3 Convex Functions . . . . .	44

3.3.1	Definition . . . . .	44
3.3.2	Basic Properties . . . . .	45
3.3.3	Examples . . . . .	45
3.3.4	$\alpha$ - sublevel sets . . . . .	47
3.3.5	Checking convexity using derivatives . . . . .	49
3.3.6	Conclusion . . . . .	50
3.4	Matrix Diagonalization and Eigen Decomposition . . . . .	51
3.4.1	Eigenvalues and Eigenvectors . . . . .	51
3.4.2	Orthogonal and Orthonormal Bases . . . . .	52
3.4.3	Normalizing Eigenvectors . . . . .	53
3.5	Singular Value Decomposition (SVD) . . . . .	54
3.5.1	Definition of SVD . . . . .	54
3.5.2	Origin of the Term Singular Value Decomposition . . . . .	55
3.5.3	SVD in Image Compression . . . . .	56
3.5.4	Conclusion . . . . .	57
3.6	Principal Component Analysis (PCA) . . . . .	58
3.6.1	Introduction . . . . .	58
3.6.2	Matrix 2-Norm . . . . .	59
3.6.3	Vector Representation in Different Bases . . . . .	60
3.6.4	Trace . . . . .	61
3.6.5	Expectation and Covariance Matrix . . . . .	62
3.6.6	Univariate Data . . . . .	62
3.6.7	Multivariate Data . . . . .	62
3.6.8	Principal Component Analysis . . . . .	62
3.6.9	Relationship between PCA and SVD . . . . .	68
3.6.10	SVD for the Best Low-Rank Approximation Problem . . . . .	68
3.6.11	The Idea of PCA . . . . .	69
3.6.12	Relationship between PCA and SVD . . . . .	69
3.6.13	How to Choose the Dimension of the New Data . . . . .	69
3.6.14	Notes on PCA in Real-World Applications . . . . .	70
3.7	Data Dimension Greater than the Number of Data Points . . . . .	71
3.8	Large-Scale Problems . . . . .	72

3.9	Likelihood Function and Maximum Likelihood Estimation . . . . .	73
3.9.1	Likelihood Function . . . . .	73
3.9.2	Log-Likelihood Function . . . . .	73
3.9.3	Maximum Likelihood Estimation . . . . .	73
3.9.4	Optimization in Logistic Regression . . . . .	74
3.10	Radial Basis Functions (RBFs) . . . . .	75
3.10.1	Overview . . . . .	75
3.10.2	Commonly Used Radial Basis Functions . . . . .	75
3.10.3	Approximation Using Radial Basis Functions . . . . .	78
3.10.4	Radial Basis Function Networks . . . . .	78
3.10.5	Radial Basis Functions for Solving PDEs . . . . .	78
3.10.6	Conclusion . . . . .	79
3.11	Sigmoid Functions . . . . .	79
3.11.1	Applications . . . . .	80
3.11.2	Comparison with Other Sigmoid Functions . . . . .	80
3.11.3	Pros of Using the Sigmoid Function . . . . .	82
3.11.4	Cons of Using the Sigmoid Function . . . . .	82
3.12	Gradient Descent . . . . .	84
3.12.1	Introduction . . . . .	84
3.12.2	Gradient Descent for Single Variable Functions . . . . .	85
3.12.3	Simple Example with Python . . . . .	86
3.12.4	Different Starting Points . . . . .	86
3.12.5	Different Learning Rates . . . . .	87
3.12.6	Algorithm Steps . . . . .	88
3.12.7	Types of Gradient Descent . . . . .	88
3.12.8	Applications Beyond Machine Learning . . . . .	89
3.12.9	Advantages and Disadvantages . . . . .	89
3.13	Stochastic Gradient Descent . . . . .	90
3.13.1	How Stochastic Gradient Descent Works . . . . .	90
3.13.2	Challenges of Stochastic Gradient Descent . . . . .	90
3.13.3	Methods to Improve Stochastic Gradient Descent Performance and Avoid Local Minimum . . . . .	91

3.14	Gradient Descent for Multi-variable Functions . . . . .	93
3.14.1	Back to Linear Regression . . . . .	93
3.14.2	Example in Python and Some Programming Notes . . . . .	93
3.14.3	Checking the Derivative . . . . .	94
3.14.4	Geometric Explanation . . . . .	95
3.14.5	Analytic Explanation . . . . .	95
3.14.6	For Multi-variable Functions . . . . .	95
3.14.7	Level Sets . . . . .	97
3.15	Optimization Problems . . . . .	98
3.16	Duality . . . . .	99
3.16.1	Lagrange Multiplier Method . . . . .	99
3.16.2	The Lagrange Dual Function . . . . .	100
3.16.3	The Lagrange Dual Problem . . . . .	102
3.16.4	Optimality Conditions . . . . .	104
<b>4</b>	<b>Basic Machine Learning</b>	<b>107</b>
4.1	Dataset . . . . .	107
4.1.1	Definition of a Dataset . . . . .	107
4.1.2	Properties of a Dataset . . . . .	107
4.1.3	Classical Datasets in Statistical Literature . . . . .	108
4.1.4	Imbalanced Datasets . . . . .	109
4.2	Synthetic Minority Over-sampling Technique (SMOTE) . . . . .	112
4.2.1	Introduction to Class Imbalance . . . . .	112
4.2.2	Addressing Class Imbalance: Traditional Over-sampling Techniques .	112
4.2.3	What is SMOTE? . . . . .	113
4.2.4	How Does SMOTE Work? . . . . .	113
4.2.5	Advantages and Limitations of SMOTE . . . . .	113
4.2.6	Implementation of SMOTE in Python . . . . .	114
4.2.7	Extensions of SMOTE . . . . .	116
4.2.8	Conclusion . . . . .	118
4.2.9	Applications of SMOTE . . . . .	118
4.3	Bayesian Modeling and Inference . . . . .	119

4.3.1	Bayesian Models . . . . .	119
4.3.2	Bayesian Inference . . . . .	119
4.3.3	Interpretation . . . . .	121
4.3.4	Evaluating the Posterior . . . . .	122
4.4	Entropy . . . . .	124
4.4.1	Overview . . . . .	124
4.4.2	Surprisal and Self-Information . . . . .	125
4.4.3	Entropy as Expected Information . . . . .	125
4.4.4	Shannon Entropy . . . . .	125
4.4.5	Types of Entropy in Information Theory . . . . .	126
4.4.6	Mutual Information . . . . .	127
4.4.7	KL Divergence . . . . .	127
4.4.7.1	The Entropy of Our Distribution . . . . .	127
4.4.7.2	Measuring Information Lost using Kullback-Leibler Divergence	128
4.5	Logistic Regression . . . . .	131
4.5.1	Logistic function . . . . .	131
4.5.2	Decision boundary . . . . .	132
4.5.3	Cross-Entropy Loss function . . . . .	132
4.5.4	Cost function . . . . .	133
4.5.5	Gradient descent . . . . .	133
4.5.6	Learning . . . . .	136
4.6	Decision Tree . . . . .	138
4.6.1	Overview . . . . .	138
4.6.2	Working Mechanism of Decision Trees . . . . .	138
4.6.3	Attribute Selection Measures . . . . .	139
4.6.4	Information Gain . . . . .	139
4.6.5	Gini Index . . . . .	139
4.6.6	Comparison of Information Gain and Gini Index . . . . .	140
4.6.7	One-hot encoding of categorical features . . . . .	140
4.6.8	Continuous valued features . . . . .	141
4.6.9	Regression Trees . . . . .	141
4.6.10	Learning Process . . . . .	141

4.7	Random Forest . . . . .	142
4.7.1	Tree ensemble . . . . .	142
4.7.2	Sampling with replacement train set . . . . .	142
4.7.3	Randomizing the feature choice . . . . .	142
4.7.4	Learning Process . . . . .	142
4.8	Noise and Outliers in Data Analysis . . . . .	143
4.8.1	Definitions . . . . .	143
4.8.2	Handling Noise and Outliers . . . . .	144
4.8.3	Comparison of Noise and Outliers . . . . .	146
4.9	Evaluation metrics . . . . .	147
4.9.1	Confusion Matrix and Fundamental Metrics . . . . .	147
4.9.2	Receiver Operating Characteristic (ROC) curve . . . . .	150
4.9.3	Type I Error vs Type II Error: An Inevitable Tradeoff . . . . .	156
4.9.4	Impossibility Results for Reliable Detection of AI-Generated Text . . . . .	157
4.10	Sampling Methods in Machine Learning . . . . .	161
4.10.1	Probability Sampling . . . . .	161
4.10.2	Non-Probability Sampling . . . . .	163
4.10.3	Sampling with and without Replacement . . . . .	164
4.10.4	Feature Randomization . . . . .	164
4.10.5	Feature Flagging . . . . .	165
4.11	Perceptron Learning Algorithm . . . . .	166
4.11.1	Problem Statement . . . . .	166
4.11.2	Loss Function Construction . . . . .	168
4.11.3	Summary . . . . .	169
<b>5</b>	<b>Support Vector Machine</b>	<b>171</b>
5.1	Support Vector Machine . . . . .	171
5.1.1	Introduction . . . . .	171
5.1.2	Constructing the Optimization Problem for SVM . . . . .	174
5.1.3	The Dual Problem for SVM. Origin of the name "Support Vector" . . . . .	176
5.1.4	SVM in Python . . . . .	180
5.1.5	Summary and Discussion . . . . .	183

5.2	Soft Margin SVM . . . . .	184
5.2.1	Mathematical Analysis . . . . .	185
5.2.2	Lagrange Dual Problem . . . . .	187
5.3	Kernel SVM . . . . .	190
5.3.1	Introduction . . . . .	190
5.3.2	Mathematical Analysis . . . . .	193
5.3.3	Properties of the Kernel Function . . . . .	195
5.3.4	Common Kernel Functions . . . . .	195
5.3.5	Summary . . . . .	196
<b>6</b>	<b>Ensemble Methods</b>	<b>197</b>
6.1	Weak and Strong Learners . . . . .	197
6.1.1	Weak Learners . . . . .	197
6.1.2	Strong Learners . . . . .	197
6.1.3	Weak vs. Strong Learners in Ensemble Learning . . . . .	198
6.2	Ensemble Learning . . . . .	200
6.2.1	History of Ensemble Learning . . . . .	200
6.2.2	Overview . . . . .	200
6.2.3	Types of Ensemble . . . . .	201
6.2.3.1	Bagging . . . . .	201
6.2.3.2	Boosting . . . . .	201
6.2.3.3	Stacking . . . . .	202
6.2.3.4	Voting . . . . .	202
6.2.3.5	Bayes Optimal Classifier . . . . .	202
6.2.3.6	Bayesian Model Averaging . . . . .	203
6.2.3.7	Bayesian Model Combination . . . . .	204
6.2.3.8	Amended Cross-Entropy Cost . . . . .	204
6.3	Voting . . . . .	206
6.3.1	Types of Ensemble Voting Methods . . . . .	206
6.3.2	Theoretical Advantages of Ensemble Voting . . . . .	207
6.3.3	Applications of Ensemble Voting in Machine Learning . . . . .	207
6.3.4	Assumptions and Limitations of Ensemble Voting . . . . .	208

6.4	Bagging . . . . .	209
6.4.1	History of Ensemble Bagging . . . . .	209
6.4.2	Technique . . . . .	209
6.4.3	Datasets in Bagging . . . . .	210
6.4.4	Classification Algorithm Process using Bootstrap Sampling . . . . .	211
6.4.5	Advantages and Disadvantages of Bagging . . . . .	212
6.5	Sequential Learning . . . . .	214
6.5.1	Concept of Sequential Learning and Sequential Models . . . . .	214
6.5.2	Sequential Data . . . . .	214
6.5.3	Types of Sequential Models . . . . .	214
6.5.3.1	Recurrent Neural Networks (RNNs) . . . . .	214
6.5.3.2	Long Short-Term Memory Networks (LSTMs) . . . . .	215
6.5.3.3	Autoencoders . . . . .	218
6.5.3.4	Sequence-to-Sequence (Seq2Seq) . . . . .	218
6.5.4	Applications of Sequential Models . . . . .	218
6.6	Boosting . . . . .	219
6.6.1	History of Boosting . . . . .	219
6.6.2	Sequential Learning . . . . .	219
6.6.3	Weighted Training Instances in Machine Learning . . . . .	220
6.6.4	How Boosting Works . . . . .	221
6.6.5	Boosting Algorithms . . . . .	222
6.6.5.1	Adaptive Boosting (AdaBoost) . . . . .	222
6.6.5.2	Gradient Boosting . . . . .	223
6.6.6	Applications of Boosting . . . . .	224
6.6.7	Challenges and Considerations . . . . .	224
6.6.8	Application of Boosting in Detecting AI-Generated Text . . . . .	225
6.7	Gradient Boosting . . . . .	226
6.7.1	Overview and History . . . . .	226
6.7.2	Loss Function . . . . .	226
6.7.2.1	Regression . . . . .	226
6.7.2.2	Classification . . . . .	228
6.7.3	Additive Model . . . . .	229

6.7.4	Gradient Descent in Gradient Boosting . . . . .	231
6.7.5	Regularization Techniques . . . . .	232
6.7.5.1	Shrinkage . . . . .	233
6.7.5.2	Stochastic Gradient Boosting . . . . .	233
6.7.5.3	Number of Observations in Leaves . . . . .	233
6.7.5.4	Complexity Penalty . . . . .	234
6.7.6	Feature Importance and Interpretability . . . . .	234
6.7.7	Comparison between AdaBoost and Gradient Boosting . . . . .	234
6.7.8	Variants of Gradient Boosting . . . . .	234
6.7.8.1	LightGBM . . . . .	235
6.7.8.2	CatBoost . . . . .	235
6.7.8.3	XGBoost . . . . .	236
6.7.9	Applications of Gradient Boosting . . . . .	236
6.7.10	Disadvantages of Gradient Boosting . . . . .	237
6.8	Extreme Gradient Boosting (XGBoost) . . . . .	238
6.8.1	Classification and Regression Trees (CART) . . . . .	238
6.8.1.1	Overview of CART in XGBoost . . . . .	238
6.8.1.2	Structure and Function of CART . . . . .	238
6.8.1.3	Types of Trees in CART . . . . .	239
6.8.1.4	Splitting Criteria in CART . . . . .	239
6.8.1.5	Algorithm and Steps for CART . . . . .	239
6.8.1.6	Pruning in CART . . . . .	240
6.8.1.7	Gini Impurity in CART . . . . .	241
6.8.1.8	Exact Greedy Algorithm . . . . .	241
6.8.1.9	Approximate Algorithm . . . . .	242
6.8.2	Construction of Candidate Splits . . . . .	242
6.8.2.1	Weighted Quantile Sketch . . . . .	243
6.8.2.2	Histogram-based Split Finding . . . . .	244
6.8.2.3	Split Gain Calculation . . . . .	245
6.8.2.4	Advantages and Applications . . . . .	245
6.8.3	Objective Function . . . . .	246
6.8.3.1	Loss Function . . . . .	246

6.8.3.2	Unregularized XGBoost Algorithm . . . . .	246
6.8.3.3	Regularization Term . . . . .	248
6.8.4	Sparsity-Aware Splitting Algorithm . . . . .	248
6.8.4.1	Objective of Sparsity-Aware Splitting . . . . .	248
6.8.4.2	Pseudocode of Sparsity-Aware Splitting . . . . .	248
6.8.4.3	Purpose and Advantages . . . . .	250
6.8.5	Second-order Approximation . . . . .	250
6.8.5.1	Taylor Expansion . . . . .	250
6.8.5.2	Regularized Learning Objective . . . . .	251
6.8.5.3	Second-order Approximation (Taylor Expansion) . . . . .	251
6.8.5.4	Gain from a Split . . . . .	252
6.8.5.5	Advantages of the Second-order Approximation in XGBoost	252
6.8.6	Hyperparameters . . . . .	252
6.8.7	Column Sampling . . . . .	255
6.8.7.1	Definition . . . . .	255
6.8.7.2	Use Case . . . . .	255
6.8.7.3	Purpose . . . . .	256
6.8.8	Parallelization . . . . .	256
6.8.8.1	Building Trees Level-wise . . . . .	256
6.8.8.2	Parallel Computations . . . . .	257
6.8.9	Scalability . . . . .	257
6.8.9.1	Out-of-core Computing . . . . .	257
6.8.9.2	Distributed Computing . . . . .	258
6.9	Distinguishing between ensembling, hybrid, mixing, and mutating . . . . .	259
6.9.1	Ensembling . . . . .	259
6.9.2	Hybrid Models . . . . .	259
6.9.3	Mixing . . . . .	259
6.9.4	Mutating . . . . .	260
<b>7</b>	<b>Deep Learning</b>	<b>261</b>
7.1	Neural Networks Notations . . . . .	261
7.2	Activation Functions . . . . .	262

7.2.1	Introduction and Historical Background . . . . .	262
7.2.2	Why Activation Functions Are Needed . . . . .	262
7.2.3	Importance of Non-Linearity . . . . .	263
7.2.4	Key Mathematical Properties of Activation Functions . . . . .	263
7.2.5	Types of Activation Functions . . . . .	264
7.2.5.1	Ridge Activation Functions . . . . .	264
7.2.5.2	Radial Activation Functions . . . . .	273
7.2.5.3	Fold Activation Functions . . . . .	275
7.2.6	Conclusion . . . . .	278
7.3	Artificial Neural Networks . . . . .	280
7.3.1	Introduction to Artificial Neural Networks . . . . .	280
7.3.2	Neurons . . . . .	280
7.3.2.1	Structure of a Neuron . . . . .	280
7.3.2.2	Biological Inspiration . . . . .	280
7.3.2.3	Artificial Neurons vs. Biological Neurons . . . . .	281
7.3.2.4	Learning and Adaptation . . . . .	281
7.3.3	Basic Structure of Neural Networks . . . . .	281
7.3.4	Feedforward . . . . .	285
7.3.5	Backpropagation . . . . .	287
7.3.5.1	Perceptron Limitation: The XOR problem . . . . .	287
7.3.5.2	The Backpropagation Process . . . . .	288
7.3.6	Types of Neural Network Architectures . . . . .	290
7.3.7	Training Neural Networks . . . . .	290
7.3.8	Advanced Optimization Techniques . . . . .	290
7.3.9	Applications of Neural Networks . . . . .	291
7.3.10	Future Directions and Challenges . . . . .	291
7.4	Feed-Forward Neural Networks . . . . .	292
7.4.1	Structure of Feed-Forward Neural Networks . . . . .	292
7.4.2	Mathematics of Feed-Forward Neural Networks . . . . .	292
7.4.3	Activation Functions in Feed-Forward Neural Networks . . . . .	293
7.4.4	Training Feed-Forward Neural Networks . . . . .	293
7.4.5	Applications of Feed-Forward Neural Networks . . . . .	293

7.4.6	Limitations of Feed-Forward Neural Networks . . . . .	294
7.4.7	Summary . . . . .	294
7.5	Stochastic Neural Networks . . . . .	295
7.5.1	Introduction to Stochasticity in Neural Networks . . . . .	295
7.5.2	Advantages of Stochastic Neural Networks . . . . .	295
7.5.3	Training Methods for Stochastic Neural Networks . . . . .	295
7.5.4	Applications of Stochastic Neural Networks . . . . .	296
7.5.5	Challenges and Future Directions . . . . .	296
7.5.6	Bayesian Neural Networks . . . . .	297
7.5.7	Benefits of Bayesian Neural Networks . . . . .	297
7.5.8	Challenges and Limitations of Bayesian Neural Networks . . . . .	297
7.5.9	Applications of Bayesian Neural Networks . . . . .	298
7.6	Convolutional Neural Networks . . . . .	299
7.6.1	Architecture of Convolutional Neural Networks . . . . .	299
7.6.2	Convolutional Operation . . . . .	299
7.6.3	Padding . . . . .	300
7.6.4	Stride . . . . .	301
7.6.5	1D Convolution in NLP . . . . .	304
7.6.6	Popular kernels . . . . .	305
7.6.7	Pooling Layers . . . . .	306
7.6.8	Engineering with Convolutional Operations . . . . .	307
7.6.9	Fully Connected Layers . . . . .	309
7.6.10	Training Convolutional Neural Networks . . . . .	309
7.6.11	Applications of Convolutional Neural Networks . . . . .	310
7.6.12	Challenges and Limitations . . . . .	310
7.6.13	Summary . . . . .	310
7.7	Recurrent Neural Networks . . . . .	311
7.7.1	Architecture of Recurrent Neural Networks . . . . .	311
7.7.2	Training Recurrent Neural Networks . . . . .	312
7.7.3	The Problem of Long-Term Dependencies . . . . .	312
7.7.3.1	Short-Term Dependencies . . . . .	312
7.7.3.2	Long-Term Dependencies . . . . .	313

7.7.3.3	Why Do RNNs Struggle?	313
7.7.3.4	Overcoming the Problem	313
7.7.4	Long Short-Term Memory (LSTM) Networks	314
7.7.5	Applications of Recurrent Neural Networks	314
7.7.6	Challenges and Limitations	315
7.7.7	Gated Recurrent Units (GRUs)	315
7.7.8	Summary	315
7.7.9	Distinguishing between Recurrent Neural Networks and Recursive Neural Networks	315
7.8	Long Short-Term Memory Networks	318
7.8.1	Architecture of LSTM	318
7.8.1.1	Memory Cell	318
7.8.1.2	Gates in LSTM	318
7.8.1.3	Updating the Memory Cell	320
7.8.2	Advantages of LSTM	321
7.8.3	Applications of LSTM Networks	321
7.8.4	Conclusion	322
7.9	Gated Recurrent Unit	323
7.9.1	Architecture of GRU	323
7.9.1.1	Gates in GRU	323
7.9.1.2	Updating the Hidden State	323
7.9.2	Advantages of GRU	324
7.9.3	Applications of GRU	324
7.9.4	Comparison to LSTM	324
7.9.5	Conclusion	325
7.10	Bidirectional Long Short-Term Memory Networks	326
7.10.1	Architecture of BiLSTM	326
7.10.2	Advantages of BiLSTM	326
7.10.3	Applications of BiLSTM Networks	327
7.10.4	Comparison with Unidirectional LSTM	327
7.10.5	LSTM Equations	327
7.10.6	Bidirectional Processing	328

7.10.7	Computational Complexity of BiLSTM . . . . .	328
7.10.8	Gradient Flow and Training . . . . .	329
7.10.9	Applications of BiLSTM Networks in Sequence Modeling . . . . .	329
7.10.10	BiLSTM Algorithm . . . . .	329
7.11	Fundamental of Embeddings . . . . .	331
7.11.1	What is Embedding? . . . . .	331
7.11.2	How Embedding Works . . . . .	331
7.11.2.1	Word2Vec Example . . . . .	331
7.11.2.2	Recommendation Embedding . . . . .	332
7.11.3	Types of Embeddings . . . . .	333
7.11.4	How Embeddings Are Created . . . . .	335
7.12	One-Hot Encoding for Word Embeddings . . . . .	337
7.12.1	Definition of One-Hot Vector for Words . . . . .	337
7.12.2	One-Hot Encoding Process . . . . .	337
7.12.3	One-Hot Encoding in Python with Sklearn . . . . .	337
7.12.4	Conclusion . . . . .	338
7.13	Pre-trained Word Embeddings: Word2Vec and GloVe . . . . .	340
7.13.1	Introduction . . . . .	340
7.13.2	Word2Vec . . . . .	340
7.13.2.1	History . . . . .	340
7.13.2.2	CBOW (Continuous Bag of Words) . . . . .	341
7.13.2.3	Skip-gram Model . . . . .	343
7.13.2.4	Parameterization of Word2Vec . . . . .	349
7.13.2.5	Extensions to Word2Vec . . . . .	349
7.13.3	GloVe (Global Vectors for Word Representation) . . . . .	351
7.13.3.1	Word Counting . . . . .	351
7.13.3.2	Probabilistic Modelling . . . . .	352
7.13.3.3	Logistic Regression . . . . .	353
7.13.3.4	Use . . . . .	354
7.13.3.5	Nearest Neighbors . . . . .	354
7.13.3.6	Linear Substructures . . . . .	354
7.13.3.7	Training . . . . .	355

7.13.4 Differences in the Properties of Word2Vec and GloVe . . . . .	357
7.13.4.1 How is the Word2Vec Model Trained? . . . . .	357
7.13.4.2 How is the GloVe Model Trained? . . . . .	358
7.13.5 Conclusion . . . . .	358
7.14 Basic Attention . . . . .	359
7.14.1 Mechanics of Basic Attention . . . . .	359
7.14.2 Applications of Basic Attention in Sequential Tasks . . . . .	359
7.14.3 Attention Variants . . . . .	360
7.14.4 Benefits of Basic Attention . . . . .	360
7.15 Self-Attention Mechanism: Capturing Contextual Dependencies within Sequences . . . . .	361
7.15.1 Motivation for Self-Attention . . . . .	361
7.15.1.1 Computational Complexity Per Layer . . . . .	361
7.15.1.2 Parallelization Potential . . . . .	361
7.15.1.3 Path Length for Long-Range Dependencies . . . . .	362
7.15.2 Mechanics of Self-Attention . . . . .	362
7.15.3 Applications in Transformer Models . . . . .	363
7.15.4 Benefits of Self-Attention . . . . .	363
7.15.5 Challenges and Considerations . . . . .	363
7.15.6 Comparison with Convolutional Layers . . . . .	363
7.16 Scaled Dot-Product Attention . . . . .	364
7.16.1 Mathematical Formulation of Scaled Dot-Product Attention . . . . .	364
7.16.2 Softmax and Output Computation . . . . .	365
7.17 Multi-Head Attention . . . . .	366
7.17.1 Mechanism of Multi-Head Attention . . . . .	366
7.17.2 Concatenation and Linear Transformation . . . . .	367
7.17.3 Advantages of Multi-Head Attention . . . . .	367
7.17.4 Application in Transformer Models . . . . .	367
7.18 Softmax with Temperature . . . . .	368
7.18.1 Standard Softmax Function . . . . .	368
7.18.2 Softmax with Temperature . . . . .	368
7.18.3 Applications of Softmax with Temperature . . . . .	369

7.19 Transformer Architecture . . . . .	370
7.19.1 Introduction to Transformers . . . . .	370
7.19.1.1 Motivation and Design Philosophy . . . . .	370
7.19.1.2 Self-Attention Mechanism . . . . .	370
7.19.1.3 Parallel Processing and Scalability . . . . .	370
7.19.1.4 Impact and Applications . . . . .	371
7.19.2 Feed-Forward Networks . . . . .	371
7.19.2.1 Architecture of Feed-Forward Networks . . . . .	371
7.19.2.2 Role in Transformer Blocks . . . . .	371
7.19.2.3 Advantages of Position-Wise Feed-Forward Networks . . . . .	372
7.19.3 Introduction of Encoder and Decoder . . . . .	372
7.19.3.1 Encoder . . . . .	373
7.19.3.2 Decoder . . . . .	375
7.19.3.3 Training and Optimization . . . . .	377
7.19.3.4 Issues with Encoder-Decoder Architectures . . . . .	378
7.19.3.5 Tokenization and Representation . . . . .	378
7.19.3.6 Positional Encoding . . . . .	379
7.20 Autoencoder . . . . .	380
7.20.1 Applications of Autoencoders . . . . .	381
7.20.2 Practical Example: Using Autoencoders for Text Denoising . . . . .	382
7.20.3 Denoising Autoencoder . . . . .	383
7.20.4 Sparse Autoencoder . . . . .	384
7.20.5 $k$ -Sparse Autoencoder . . . . .	385
7.20.6 Variational Autoencoder (VAE) . . . . .	386
7.20.6.1 Loss Function: ELBO . . . . .	387
7.20.6.2 Reparameterization Trick . . . . .	390
7.20.6.3 Beta-VAE . . . . .	390
7.21 Practical NLP Tasks and Applications . . . . .	393
7.21.1 Sentence-Level Tasks . . . . .	393
7.21.2 Token-Level Tasks . . . . .	394
7.21.3 Downstream Tasks . . . . .	395

<b>8</b>	<b>Code Engineering</b>	<b>397</b>
8.1	Datasets . . . . .	397
8.1.1	Detect AI Generated Text . . . . .	397
8.1.2	DaiGT - Proper Train . . . . .	398
8.1.3	LLM - Detect AI Generated Text Dataset . . . . .	400
8.1.4	Overview . . . . .	401
8.2	Data Preprocessing . . . . .	402
8.2.1	Datasets . . . . .	402
8.2.1.1	Data Collection Process . . . . .	402
8.2.1.2	Data Preprocessing and Merging . . . . .	402
8.2.1.3	Publishing the Merged Dataset on Kaggle . . . . .	403
8.2.1.4	Final Remarks . . . . .	404
8.2.2	Data Cleaning and Preparation . . . . .	404
8.2.2.1	Importing Libraries and Initial Setup . . . . .	404
8.2.2.2	Text Cleaning . . . . .	404
8.2.2.3	Stopword Removal . . . . .	405
8.2.2.4	Tokenization . . . . .	405
8.2.3	Feature Engineering . . . . .	406
8.2.4	Data Splitting . . . . .	406
8.3	Analyze the Training Process of Models . . . . .	407
8.3.1	Model: Logistic Regression . . . . .	407
8.3.1.1	Introduction . . . . .	407
8.3.1.2	Training Configuration . . . . .	407
8.3.1.3	Training and Evaluation Results and Discussion . . . . .	407
8.3.1.4	Observations . . . . .	408
8.3.1.5	Test Results . . . . .	409
8.3.1.6	Conclusion . . . . .	409
8.3.2	Model XGBoost . . . . .	410
8.3.2.1	Introduction . . . . .	410
8.3.2.2	Training Configuration . . . . .	410
8.3.2.3	Training and Evaluation Results and Discussion . . . . .	410
8.3.2.4	Test Metrics . . . . .	411

8.3.2.5	Recommendations for Improvement . . . . .	412
8.3.2.6	Conclusion . . . . .	412
8.3.3	Model: Random Forest . . . . .	413
8.3.3.1	Introduction . . . . .	413
8.3.3.2	Training Configuration . . . . .	413
8.3.3.3	Metrics Evaluation . . . . .	413
8.3.3.4	Results and Discussion . . . . .	413
8.3.3.5	Test Results . . . . .	416
8.3.3.6	Conclusion . . . . .	416
8.3.4	Model: LinearSVC . . . . .	417
8.3.4.1	Introduction . . . . .	417
8.3.4.2	Training Configuration . . . . .	417
8.3.4.3	Training and Evaluation Results and Discussion . . . . .	417
8.3.4.4	Observations . . . . .	418
8.3.4.5	Test Results . . . . .	419
8.3.4.6	Conclusion . . . . .	420
8.3.5	Model: Bidirectional LSTM (a type of RNN) . . . . .	421
8.3.5.1	Introduction . . . . .	421
8.3.5.2	Training Configuration . . . . .	421
8.3.5.3	Training and Evaluation Results and Discussion . . . . .	421
8.3.5.4	Observations . . . . .	422
8.3.5.5	Test Results . . . . .	423
8.3.5.6	Conclusion . . . . .	423
8.3.6	Model: DistilBERT . . . . .	424
8.3.6.1	Introduction . . . . .	424
8.3.6.2	Training Configuration . . . . .	424
8.3.6.3	Metrics Evaluation . . . . .	424
8.3.6.4	Results and Discussion . . . . .	425
8.3.6.5	Test Results . . . . .	426
8.3.7	Model Comparison for AI-Generated Text Detection . . . . .	428
8.3.7.1	Discussion . . . . .	428
8.3.7.2	Type I and Type II Error Considerations . . . . .	429

8.3.7.3	Model Submissions to Kaggle Competition . . . . .	430
8.3.7.4	Evaluation . . . . .	432
8.3.8	Conclusion . . . . .	433
<b>9</b>	<b>Self-Reflection</b>	<b>435</b>
9.1	About Our Blog . . . . .	435
9.2	Future Developments . . . . .	435
9.3	Special Thanks . . . . .	436



# List of Figures

3.1 Examples of convex sets.	41
3.2 Examples of nonconvex sets.	42
3.3 Shapes of norm balls with different $p$ -norms.	43
3.4 Convex Function.	44
3.5 Example of Pointwise Maximum.	45
3.6 Examples of Convex One-variable Functions.	46
3.7 Examples of Concave One-variable Functions.	46
3.8 Examples of 1-norm (left) and 2-norm (right).	47
3.9 All alpha-sublevel sets are convex sets, but the function is non-convex.	48
3.10 First-order convexity check. Left: convex function, right: non-convex function.	49
3.11 SVD of matrix $\mathbf{A}$ when: $m < n$ (top), and $m > n$ (bottom). $\Sigma$ is a diagonal matrix with non-negative, descending elements. Darker red indicates larger values. White cells represent zeros in the matrix.	54
3.12 Example of SVD for images.	56
3.13 Matrix Projection for Dimensionality Reduction. The original data is the columns of the matrix $\mathbf{X}$ . The matrix $\mathbf{P} \in \mathbb{R}^{K \times D}$ projects each column of the data matrix onto a new lower-dimensional space. The new data is given in the matrix $\mathbf{C}$ .	58
3.14 Coordinate transformation in different bases.	61
3.15 The main idea of PCA: Find a new orthonormal basis such that the most important components are in the first $K$ components.	63
3.16 PCA from a statistical perspective. PCA finds a new orthonormal basis that acts as a rotation, so in this basis, certain dimensions have very low variance and can be ignored.	66
3.17 Steps of PCA	68
3.18 Gradient Descent: Introduction	84

4.1	Various plots of the multivariate data set Iris flower data set introduced by Ronald Fisher (1936) . . . . .	108
4.2	Extremely imbalanced dataset in AI-generated text detection. . . . .	110
4.3	Class Distribution Before and After Applying SMOTE . . . . .	116
4.4	Class Distribution Before and After Applying SMOTE Adasyn . . . . .	118
4.5	. . . . .	121
4.6	Entropy in General . . . . .	124
4.7	Entropy . . . . .	126
4.8	Caption . . . . .	130
4.9	Logistic function . . . . .	131
4.10	$-\log(f_{\vec{w},b}(\vec{x}^{(i)}))$ . . . . .	133
4.11	$\log(1 - f_{\vec{w},b}(\vec{x}^{(i)}))$ . . . . .	133
4.12	positive slope . . . . .	134
4.13	negative slope . . . . .	135
4.14	Stuck at a local minimum . . . . .	136
4.15	Information Gain . . . . .	140
4.16	Sampling with replacement train set . . . . .	142
4.17	Outlier . . . . .	144
4.18	Inter Quartile Range . . . . .	145
4.19	ROC Curve of Support Vector Machine vs Random Forest . . . . .	151
4.20	A Perfect ROC "Curve" . . . . .	151
4.21	A totally random ROC "Curve" . . . . .	152
4.22	Simulated ROC curve result of a random classifier . . . . .	153
4.23	Total variation distance is half the absolute area between the two curves: Half the shaded area above. . . . .	158
4.24	Simple Random Sampling . . . . .	161
4.25	Systematic Sampling . . . . .	162
4.26	Stratified Sampling . . . . .	162
4.27	Cluster Sampling . . . . .	162
4.28	Non-Probability Sampling . . . . .	163
4.29	Sampling with and without Replacement . . . . .	164
4.30	The Perceptron Problem . . . . .	166

4.31	Equation of the decision boundary . . . . .	167
4.32	Arbitrary decision boundary with misclassified points circled. . . . .	168
5.1	Separating hyperplanes for linearly separable classes. . . . .	172
5.2	The margin of both classes is equal and as wide as possible. . . . .	173
5.3	Analyzing the SVM problem . . . . .	174
5.4	Points closest to the separating plane of the two classes are circled. . . . .	175
5.5	Illustration of the solution obtained by SVM. . . . .	182
5.6	a: Noise within data; b: Almost linearly separable data . . . . .	184
5.7	. . . . .	185
5.8	Failure of the soft margin approach . . . . .	190
5.9	Data when transformed into a new space . . . . .	191
5.10	Classification after transforming data into the new space . . . . .	191
6.1	Weak and Strong Learners . . . . .	198
6.2	Soft and Hard Voting Examples . . . . .	207
6.3	An illustration for the concept of bootstrap aggregating . . . . .	210
6.4	Bootstrap Aggregating . . . . .	210
6.5	Bagging Algorithm for Classification . . . . .	212
6.6	Process Sequences . . . . .	215
6.7	Process Sequences . . . . .	216
6.8	Boosting Demonstration . . . . .	220
6.9	Boosting Algorithms . . . . .	222
6.10	Architecture of Gradient Boosting . . . . .	227
6.11	Architecture of Gradient Boosting . . . . .	228
6.12	Generalized Additive Models . . . . .	230
6.13	CART . . . . .	238
6.14	Pruning in CART . . . . .	240
6.15	Gini Impurity Calculation . . . . .	241
7.1	XOR as a neural network . . . . .	288
7.2	A pooling layer with pooling size 2x2 . . . . .	307
7.3	An unrolled recurrent neural network . . . . .	311

7.4	Short-Term Dependencies . . . . .	312
7.5	Long-Term Dependencies . . . . .	313
7.6	LSTM architecture . . . . .	318
7.7	Forget Gate . . . . .	319
7.8	Input Gate . . . . .	319
7.9	Update memory cell state . . . . .	320
7.10	Update hidden state . . . . .	321
7.11	Continuous Bag of Words (CBOW) Model: The context words ("the", "man", "his", "son") are used to predict the target word "loves". . . . .	343
7.12	Skip-Gram Model Example: Demonstrating the generation of training samples by pairing context words with a target word from the source text. . . . .	344
7.13	Word2Vec: Skip-Gram . . . . .	346
7.14	Skip-gram Model: The target word "fox" is used to predict context words ("quick", "brown", "jumps", "over"). . . . .	348
7.15	GloVe: Nearest Neighbors Example (Stanford) . . . . .	354
7.16	GloVe: Linear Structure Example (Stanford) . . . . .	355
7.17	CBOW vs Skip-Gram . . . . .	358
7.18	Scaled Dot-Product Attention . . . . .	364
7.19	Multi-Head Attention consists of several attention layers running in parallel . . . . .	366
7.20	Transformer architecture . . . . .	372
7.21	Autoencoders architecture . . . . .	381
7.22	Illustration of Autoencoder . . . . .	383
7.23	Illustration of denoising autoencoder model architecture.n . . . . .	384
7.24	KL divergence between a Bernoulli distribution with mean $\rho = 0.25$ and a Bernoulli distribution with mean $0 \leq \hat{\rho} \leq 1$ . . . . .	385
7.25	The graphical model involved in Variational Autoencoder. Solid lines denote the generative distribution $p_{\theta}(.)$ and dashed lines denote the distribution $q_{\phi}(\mathbf{z} \mathbf{x})$ to approximate the intractable posterior $p_{\theta}(\mathbf{z} \mathbf{x})$ . . . . .	387
7.26	Forward and reversed KL divergence have different demands on how to match two distributions <sup>1</sup> . . . . .	388
7.27	Illustration of how the reparameterization trick makes the <b>z</b> sampling process trainable <sup>2</sup> . . . . .	390

<sup>1</sup><https://blog.evjang.com/2016/08/variational-bayes.html>

<sup>2</sup>Slide 12 in Kingma's NIPS 2015 workshop. [http://dpkingma.com/wordpress/wp-content/uploads/2015/12/talk\\_nips\\_workshop\\_2015.pdf](http://dpkingma.com/wordpress/wp-content/uploads/2015/12/talk_nips_workshop_2015.pdf)

7.28	Illustration of the variational autoencoder model with the multivariate Gaussian assumption.	391
7.29	Application of Natural Language Processing	393
8.1	Training Metrics over Epochs	408
8.2	XGBoost Training Metrics over Boosting Rounds	410
8.3	Learning Curves for ROC AUC of the Random Forest model	414
8.4	Learning Curves for Accuracy of the Random Forest model	414
8.5	Validation Curve for <code>n_estimators</code> in the Random Forest model	415
8.6	Learning Curves for AUC and Accuracy of the LinearSVC model	418
8.7	Validation Curve for <code>C</code> in the LinearSVC model	419
8.8	Training Metrics over Epochs for the Bidirectional LSTM RNN Model	422
8.9	Training Metrics over Epochs for the DistilBERT Model	425
8.10	Kaggle competition submissions for detecting AI-generated text	431



# List of Tables

4.1	Degree of imbalance based on percentage of data belonging to the minority class.	109
4.2	Comparison of SMOTE Extensions	117
4.3	Comparison between Information Gain and Gini Index in Decision Trees	140
4.4	Comparison of Noise and Outliers in Data Analysis	146
4.5	Confusion matrix for the AI-generated text detection system.	147
4.6	Evaluation Metric Comparison	155
6.1	Difference between AdaBoost and Gradient Boosting	234
7.1	Table of Activation Functions (Part 1)	278
7.2	Table of Activation Functions (Part 2)	279
7.3	Notations used in Autoencoders	380
8.1	Comparison of AI-Generated Text Detection Models	428



# **Chapter 1**

## **Abstract**

With the advancement in capabilities of Large Language Models (LLMs), especially in education, concerns about maintaining honesty in essay writing have grown. This project aims to satisfy the need for detecting essays generated by LLMs by developing a machine learning model that can differentiate between LLM-generated content and essays written by students. Our goal is to create a reliable tool for students to maintain academic integrity, recognizing the proper use of LLMs in education and contributing to the field of LLM text detection.



# Chapter 2

## Description

The increasing use of Large Language Models (LLMs) in areas such as education, creative writing, and content generation has raised social concerns about their potential misuse to create misleading or AI-generated text. Because LLMs produce text that closely mimic human writing, it has become harder to differentiate between content written by people and that by machines. This is a serious problem in such fields as journalism, social media, and specially education, where accuracy and authenticity are needed.

This project is to develop a Machine Learning (ML) model that can effectively distinguish between essays generated by LLMs and those written by students. Our primary goal is to provide educators with a reliable tool to uphold academic integrity while recognizing the proper use of LLMs as educational helping tools. It is important to know that when used responsibly, LLMs can be beneficial in education, helping students refine their writing, deepen their understanding of various subjects, and foster creativity. However, a balance must be struck between leveraging these advantages and preventing misuse.

Given the limitations of this project, our focus will be on learning the essential theories and concepts behind LLM-generated text detection. We will explore how to apply these ideas in practice and reimplement some established methods in this field. The goal is to build a strong understanding of the topic while also demonstrating practical applications. In doing so, this project aims to contribute to the growing area of AI-generated text detection, supporting the integrity of written communication while utilizing AI and machine learning.



# Chapter 3

## Basic Math and Linear Algebra

### 3.1 Notation

#### 3.1.1 Set notation

$\in$	is an element of
$\notin$	is not an element of
$\{x_1, x_2, \dots\}$	the set with elements $x_1, x_2, \dots$
$\{x : \dots\}$	the set of all $x$ such that ...
$n(A)$	the number of elements in set $A$
$\emptyset$	the empty set
$\mathcal{E}$	the universal set
$\cup$	the universal set (for 0607 IGCSE International Mathematics)
$A'$	the complement of the set $A$
$\mathbb{N}$	the set of natural numbers, $\{1, 2, 3, \dots\}$
$\mathbb{Z}$	the set of integers, $\{0, \pm 1, \pm 2, \pm 3, \dots\}$
$\mathbb{Q}$	the set of rational numbers, $\left\{ \frac{p}{q} : p, q \in \mathbb{Z}, q \neq 0 \right\}$
$\mathbb{R}$	the set of real numbers
$\mathbb{C}$	the set of complex numbers
$(x, y)$	the ordered pair $x, y$
$\subset$	is a subset of
$\subsetneq$	is a proper subset of
$\cup$	union
$\cap$	intersection
$[a, b]$	the closed interval $\{x \in \mathbb{R} : a \leq x \leq b\}$
$[a, b)$	the interval $\{x \in \mathbb{R} : a \leq x < b\}$
$(a, b]$	the interval $\{x \in \mathbb{R} : a < x \leq b\}$
$(a, b)$	the open interval $\{x \in \mathbb{R} : a < x < b\}$
$(S, \circ)$	the group consisting of the set $S$ with binary operation $\circ$

### 3.1.2 Miscellaneous symbols

$=$	is equal to
$\neq$	is not equal to
$\equiv$	is identical to or is congruent to
$\approx$	is approximately equal to
$\sim$	is distributed as
$\cong$	is isomorphic to
$\propto$	is proportional to
$<$	is less than
$\leq$	is less than or equal to
$>$	is greater than
$\geq$	is greater than or equal to
$\infty$	infinity
$\Rightarrow$	implies
$\Leftarrow$	is implied by
$\Leftrightarrow$	implies and is implied by (is equivalent to)

### 3.1.3 Operations

$a + b$	a plus $b$
$a - b$	a minus $b$
$a \times b, ab$	a multiplied by $b$
$a \div b, \frac{a}{b}$	$a$ divided by $b$
$\sum_{i=1}^n a_i$	$a_1 + a_2 + \dots + a_n$
$\sqrt{a}$	the non-negative square root of $a$ , for $a \in \mathbb{R}, a \geq 0$
$\sqrt[n]{a}$	the (real) $n$ th root of $a$ , for $a \in \mathbb{R}$ , where $\sqrt[n]{a} \geq 0$ for $a \geq 0$
$ a $	the modulus of $a$
$n!$	$n$ factorial
$\binom{n}{r}$	the binomial coefficient $\frac{n!}{r!(n-r)!}$ , for $n, r \in \mathbb{Z}$ and $0 \leq r \leq n$

### 3.1.4 Functions

$f(x)$	the value of the function $f$ at $x$
$f : A \rightarrow B$	$f$ is a function under which each element of set $A$ has an image in set $B$
$f : x \mapsto y$	the function $f$ maps the element $x$ to the element $y$
$f^{-1}$	the inverse function of the one-one function $f$

$gf$	the composite function of $f$ and $g$ which is defined by $gf(x) = g(f(x))$
$\lim_{x \rightarrow a} f(x)$	the limit of $f(x)$ as $x$ tends to $a$
$\Delta x, \delta x$	an increment of $x$
$\frac{dy}{dx}$	the derivative of $y$ with respect to $x$
$\frac{d^n y}{dx^n}$	the $n$ th derivative of $y$ with respect to $x$
$f'(x), f''(x), \dots, f^{(n)}(x)$	the first, second, $\dots$ , $n$ th derivatives of $f(x)$ with respect to $x$
$\int y dx$	the indefinite integral of $y$ with respect to $x$
$\int_a^b y dx$	the definite integral of $y$ with respect to $x$ between $x = a$ and $x = b$
$\dot{x}, \ddot{x}, \dots$	the first, second, $\dots$ , derivatives of $x$ with respect to $t$

### 3.1.5 Exponential and logarithmic functions

$e$	base of natural logarithms
$e^x, \exp(x)$	exponential function of $x$
$\log_a x$	logarithm to the base $a$ of $x$
$\ln x$	natural logarithm of $x$
$\lg x, \log_{10} x$	logarithm of $x$ to base 10

### 3.1.6 Circular and hyperbolic functions

$\{\sin, \cos, \tan$	the circular functions
$\{\csc^{-1}, \sec^{-1}, \cot^{-1}$	the inverse circular functions
$\{\sinh, \cosh, \tanh$	the hyperbolic functions
$\{\operatorname{cosech}, \operatorname{sech}, \operatorname{coth}$	the inverse hyperbolic functions

### 3.1.7 Complex numbers

$i$	the imaginary unit, $i^2 = -1$
$z$	a complex number, $z = x + iy = r(\cos \theta + i \sin \theta)$
$\operatorname{Re} z$	the real part of $z$ , $\operatorname{Re} z = x$
$\operatorname{Im} z$	the imaginary part of $z$ , $\operatorname{Im} z = y$

$ z $	the modulus of $z$ , $ z  = \sqrt{x^2 + y^2}$
$\arg z$	the argument of $z$ , $\arg z = \theta$ where $-\pi < \theta \leq \pi$
$z^*$	the complex conjugate of $z$ , $x - iy$

### 3.1.8 Matrices

$\mathbf{M}$	a matrix $\mathbf{M}$
$\mathbf{M}^{-1}$	the inverse of the non-singular square matrix $\mathbf{M}$
$\det \mathbf{M},  \mathbf{M} $	the determinant of the square matrix $\mathbf{M}$
$\mathbf{I}$	an identity (or unit) matrix

### 3.1.9 Vectors

$\mathbf{a}$	the vector $\mathbf{a}$
$\overrightarrow{AB}$	the vector represented in magnitude and direction by the directed line segment $AB$
$\hat{\mathbf{a}}$	a unit vector in the direction of $\mathbf{a}$
$\mathbf{i}, \mathbf{j}, \mathbf{k}$	unit vectors in the directions of the Cartesian coordinate axes
$\begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad \begin{pmatrix} x \\ y \\ z \end{pmatrix}$	the vectors $x\mathbf{i} + y\mathbf{j}$ (in 2 dimensions) and $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ (in 3 dimensions)
$ \mathbf{a} , a$	the magnitude of $\mathbf{a}$
$ \overrightarrow{AB} , AB$	the magnitude of $\overrightarrow{AB}$
$\mathbf{a} \cdot \mathbf{b}$	the scalar product of $\mathbf{a}$ and $\mathbf{b}$
$\mathbf{a} \times \mathbf{b}$	the vector product of $\mathbf{a}$ and $\mathbf{b}$

## 3.2 Convex Sets

### 3.2.1 Definition

The concept of convex sets might not be unfamiliar to you, as we have heard about convex polygons. "Convex" can be understood simply as "bulging outward" or "protruding outward." In mathematics, even a flat surface is considered convex.

**Definition 1:** A set is called a *convex set* if the line segment connecting any two points within the set lies entirely within the set.

Here are some examples of convex sets:



Example of convex sets

Figure 3.1: Examples of convex sets.

The black-bordered figures indicate that the boundaries are included, while the white-bordered figures indicate that the boundaries are not part of the set. A line or line segment is also a convex set according to the above definition.

Some real-world examples:

- Suppose there is a convex-shaped room, and if we place a sufficiently bright light bulb at any position in the room, every point in the room will be illuminated.
- If a country has a convex-shaped map, the flight path (Manhattan path) between any two cities within that country will lie entirely within its airspace. The map of Vietnam is not convex-shaped, as Cambodia is on the flight path between Hanoi and HCMC.

Below are some examples of *nonconvex sets*, which are not convex:

The first three shapes are not convex because the dashed lines contain points that are not part of the set. The fourth shape, a square without a base, is not convex because the line connecting two points at the base may include points that are not part of the set. A curve is also not a convex set, as the line connecting two points is not entirely contained within the curve.

To describe a convex set mathematically, we use:

**Definition 2:** A set  $\mathcal{C}$  is called convex if for any two points  $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{C}$ , the point  $\mathbf{x}_\theta = \theta\mathbf{x}_1 + (1 - \theta)\mathbf{x}_2$  also lies in  $\mathcal{C}$  for any  $0 \leq \theta \leq 1$ .

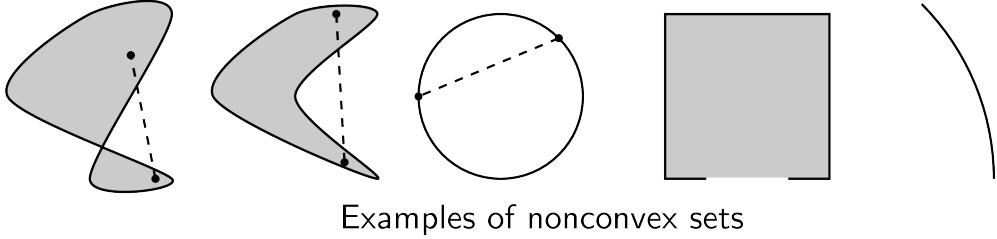


Figure 3.2: Examples of nonconvex sets.

It is clear that the set of points in the form  $\theta\mathbf{x}_1 + (1 - \theta)\mathbf{x}_2$  is the line segment connecting  $\mathbf{x}_1$  and  $\mathbf{x}_2$ .

With this definition, the entire space is a convex set because any line segment lies within the space. The empty set can also be considered a special case of a convex set.

### 3.2.2 Examples

#### Hyperplanes and halfspaces

A *hyperplane* in  $n$ -dimensional space is the set of points that satisfy the equation:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = \mathbf{a}^T\mathbf{x} = b$$

where  $b$  and  $a_i$ ,  $i = 1, 2, \dots, n$  are real numbers.

Hyperplanes are convex sets. This can be easily deduced from Definition 1. With Definition 2, we can also see this. If  $\mathbf{a}^T\mathbf{x}_1 = \mathbf{a}^T\mathbf{x}_2 = b$ , then for any  $0 \leq \theta \leq 1$ :

$$\mathbf{a}^T\mathbf{x}_\theta = \mathbf{a}^T(\theta\mathbf{x}_1 + (1 - \theta)\mathbf{x}_2) = \theta b + (1 - \theta)b = b$$

A *halfspace* in  $n$ -dimensional space is the set of points that satisfy the inequality:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = \mathbf{a}^T\mathbf{x} \leq b$$

Halfspaces are also convex sets, which can be easily seen from Definition 1 or proved using Definition 2.

#### Norm balls

*Euclidean norm balls* (circles in two-dimensional space, spheres in three-dimensional space) are the set of points defined by:

$$B(\mathbf{x}_c, r) = \left\{ \mathbf{x} \mid \|\mathbf{x} - \mathbf{x}_c\|_2 \leq r \right\} = \{ \mathbf{x}_c + r\mathbf{u} \mid \|\mathbf{u}\|_2 \leq 1 \}$$

where  $\mathbf{x}_c$  is the center of the ball and  $r$  is the radius.

The Euclidean ball is a convex set, which can be verified by applying Definition 2. Using any norm  $p \geq 1$ , we still obtain convex sets.

Below is the shape of norm balls with different values of  $p$ :

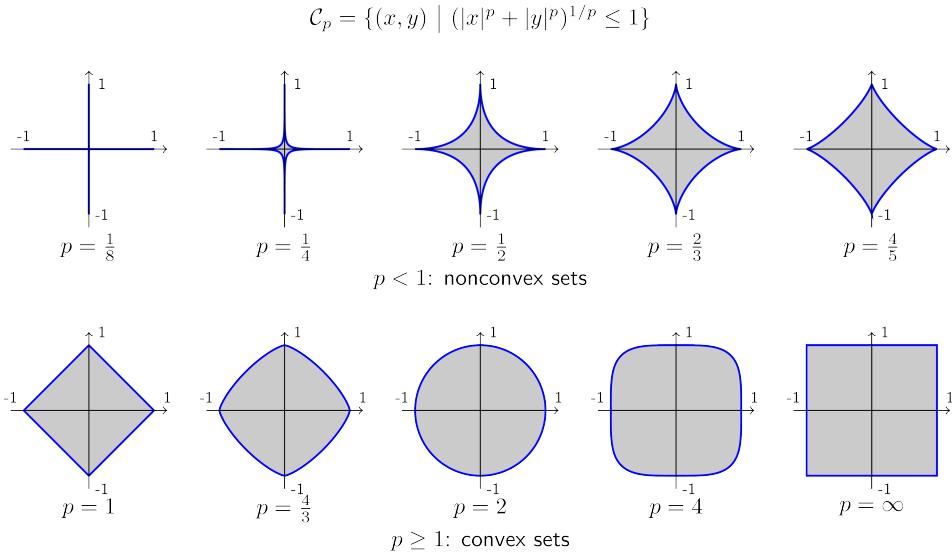


Figure 3.3: Shapes of norm balls with different  $p$ -norms.

For  $p = 1$ , the norm ball is a square. For  $p = \infty$ , the norm ball is a rhombus (diamond). As  $p$  increases, the shape of the norm ball gradually becomes a square.

### 3.2.3 Intersection of Convex Sets

The intersection of convex sets is also a convex set. This property is intuitive from Figure 3.1 (left), where the intersection of two convex sets forms another convex set.

Mathematically, if  $\mathbf{x}_1, \mathbf{x}_2$  lie in the intersection of convex sets, then the point  $\theta\mathbf{x}_1 + (1 - \theta)\mathbf{x}_2$  will also lie in that intersection. Thus, the intersection of convex sets is itself convex.

## 3.3 Convex Functions

### 3.3.1 Definition

To build an intuitive understanding, let us first examine one-dimensional functions, where the graph of the function is a curve in a plane. A function is called *convex* if its *domain* is a convex set, and for any two points on the graph, the line segment connecting them lies above or on the graph itself (refer to Figure 3.4).

The domain of a function  $f(\cdot)$  is denoted as  $\text{dom}f$ .

**Definition 3.3.1** (Convex Function). A function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is called a *convex function* if  $\text{dom}f$  is a convex set, and for all  $\mathbf{x}, \mathbf{y} \in \text{dom}f$  and  $0 \leq \theta \leq 1$ ,

$$f(\theta\mathbf{x} + (1 - \theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1 - \theta)f(\mathbf{y})$$

The condition that  $\text{dom}f$  is convex is crucial because, without it, we cannot define  $f(\theta\mathbf{x} + (1 - \theta)\mathbf{y})$ .

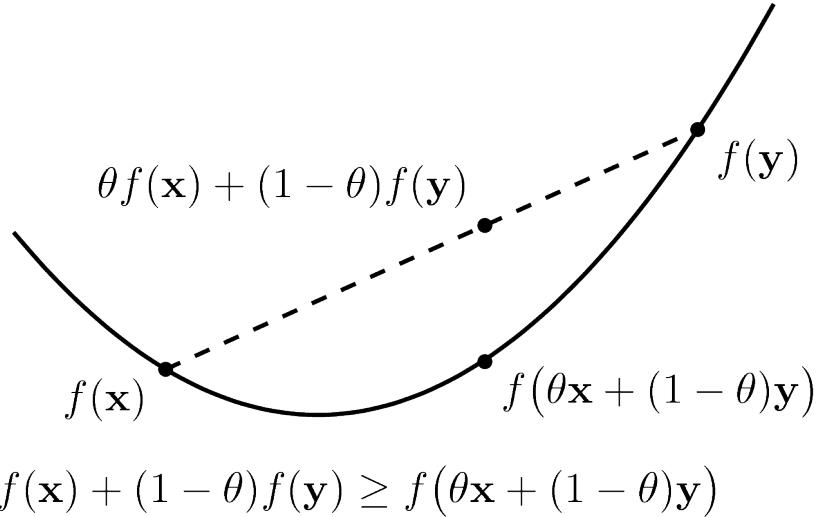


Figure 3.4: Convex Function.

A function  $f$  is called *concave* if  $-f$  is convex. A function can be neither convex nor concave. Linear functions are both convex and concave.

**Definition 3.3.2** (Strictly convex function). A function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is called *strictly convex* if  $\text{dom}f$  is convex, and for all  $\mathbf{x}, \mathbf{y} \in \text{dom}f$ ,  $\mathbf{x} \neq \mathbf{y}$ , and  $0 < \theta < 1$ ,

$$f(\theta\mathbf{x} + (1 - \theta)\mathbf{y}) < \theta f(\mathbf{x}) + (1 - \theta)f(\mathbf{y})$$

Similarly, a function is called *strictly concave* if  $-f$  is strictly convex.

An important note: *If a function is strictly convex and has an extremum, that extremum is unique and is also the global minimum.*

### 3.3.2 Basic Properties

- If  $f(\mathbf{x})$  is convex, then  $af(\mathbf{x})$  is convex if  $a > 0$ , and concave if  $a < 0$ . This follows directly from the definition.
- The sum of two convex functions is convex, with the domain being the intersection of their domains (the intersection of two convex sets is also convex).
- **Pointwise Maximum and Supremum:** If the functions  $f_1, f_2, \dots, f_m$  are convex, then  $f(\mathbf{x}) = \max\{f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_m(\mathbf{x})\}$  is also convex, where the domain is the intersection of all the domains of the functions. The maximum can also be replaced by the supremum<sup>1</sup>. This property can be proven using the definition of convexity, as illustrated by Figure 3.5.

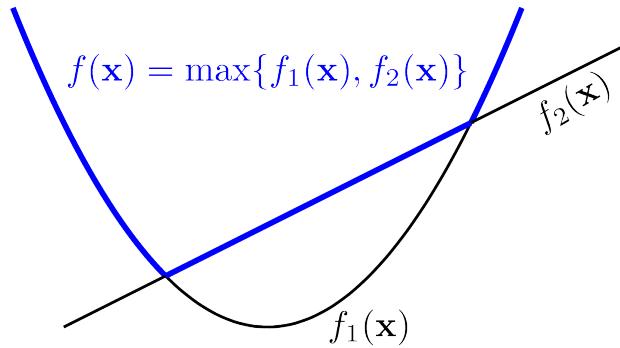


Figure 3.5: Example of Pointwise Maximum.

### 3.3.3 Examples

#### One-variable Functions

Examples of convex one-variable functions include:

- The function  $y = ax + b$  is convex because the line segment between any two points lies on the graph itself.
- The exponential function  $y = e^{ax}$  for any  $a \in \mathbb{R}$ .
- The power function  $y = x^a$  on the domain of positive real numbers, for  $a \geq 1$  or  $a \leq 0$ .
- The negative entropy function  $y = x \log x$  on the domain of positive real numbers.

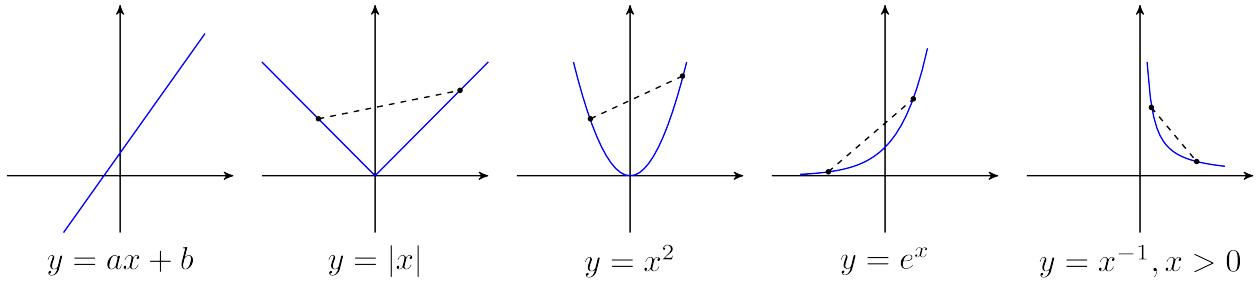


Figure 3.6: Examples of Convex One-variable Functions.

Examples of concave one-variable functions include:

- The function  $y = ax + b$  is concave since  $-y$  is convex.
- The power function  $y = x^a$  on the domain of positive real numbers for  $0 \leq a \leq 1$ .
- The logarithmic function  $y = \log(x)$  on the domain of positive real numbers.

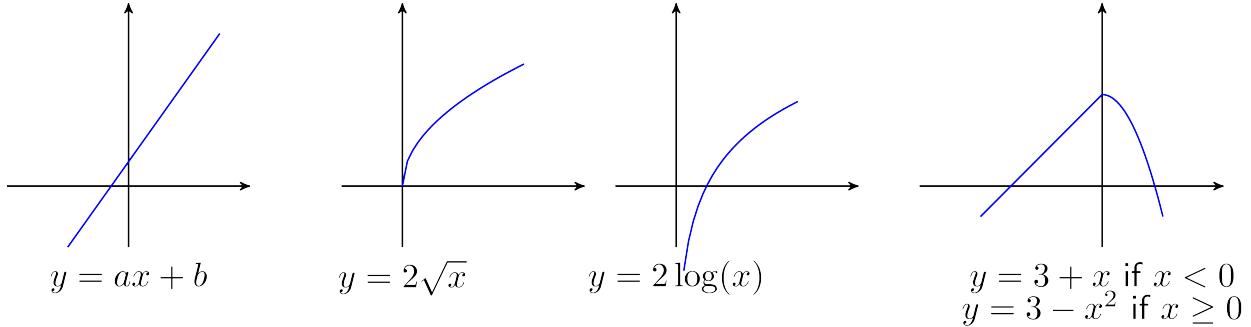


Figure 3.7: Examples of Concave One-variable Functions.

## Affine Functions

Affine functions of the form  $f(\mathbf{x}) = \mathbf{a}^T \mathbf{x} + b$  are both convex and concave.

When the variable is a matrix  $\mathbf{X}$ , affine functions take the form:

$$f(\mathbf{X}) = \text{trace}(\mathbf{A}^T \mathbf{X}) + b$$

where trace denotes the sum of the diagonal elements of a square matrix, and  $\mathbf{A}$  is a matrix of the same dimension as  $\mathbf{X}$  (ensuring matrix multiplication is well-defined).

## Quadratic Forms

A quadratic function of the form  $f(x) = ax^2 + bx + c$  is convex if  $a > 0$  and concave if  $a < 0$ .

For a vector  $\mathbf{x} = [x_1, x_2, \dots, x_n]$ , a quadratic form takes the form:

$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c$$

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Infimum\\_and\\_supremum](https://en.wikipedia.org/wiki/Infimum_and_supremum)

where  $\mathbf{A}$  is typically a symmetric matrix (i.e.,  $a_{ij} = a_{ji}$  for all  $i, j$ ), and  $\mathbf{b}$  is a matrix of the same dimension as  $\mathbf{x}$ .

If  $\mathbf{A}$  is positive semidefinite, then  $f(\mathbf{x})$  is convex. If  $\mathbf{A}$  is negative semidefinite,  $f(\mathbf{x})$  is concave.

The loss function in Linear Regression has the form:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 = \frac{1}{2} (\mathbf{y} - \mathbf{X}\mathbf{w})^T (\mathbf{y} - \mathbf{X}\mathbf{w})$$

Since  $\mathbf{X}^T \mathbf{X}$  is positive semidefinite, the loss function of Linear Regression is a convex function.

### Norms

Norms, which satisfy the three conditions of a norm, are also convex functions. Below are examples of the 1-norm (left) and 2-norm (right) with two variables:

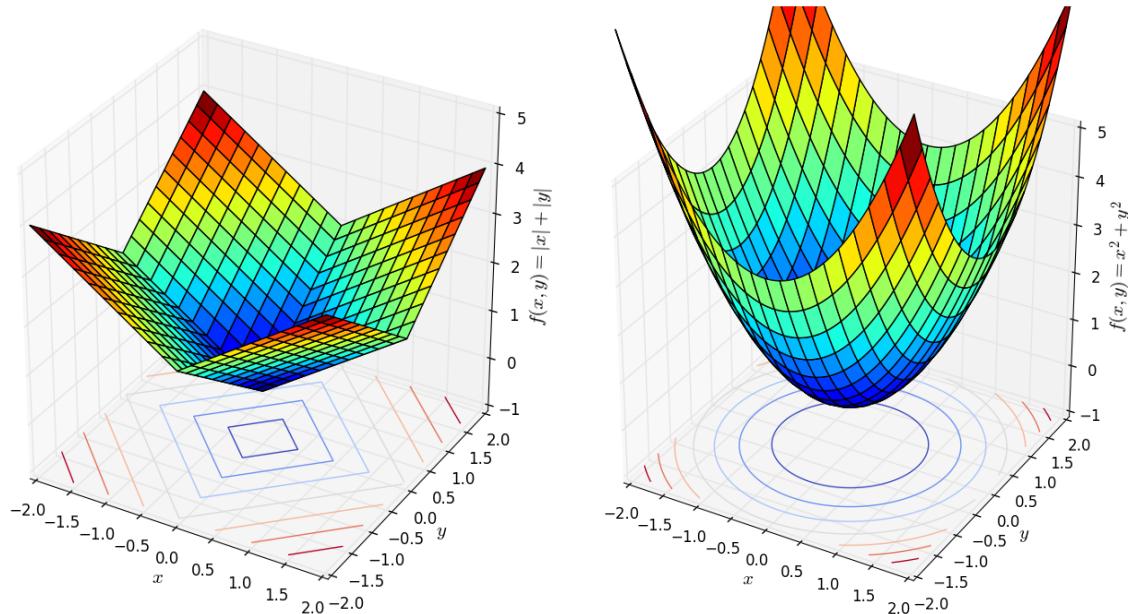


Figure 3.8: Examples of 1-norm (left) and 2-norm (right).

These surfaces exhibit a unique bottom corresponding to the origin, a characteristic of strictly convex functions.

#### 3.3.4 $\alpha$ - sublevel sets

**Definition:** An  $\alpha$ -sublevel set of a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is defined as:

$$\mathcal{C}_\alpha = \{\mathbf{x} \in \text{dom } f \mid f(\mathbf{x}) \leq \alpha\}$$

That is, the set of points in the domain of  $f$  where  $f$  takes values less than or equal to  $\alpha$ .

Referring back to Figure 3.8, the  $\alpha$ -sublevel sets correspond to the areas enclosed by the level sets.

In the bottom left image, the  $\alpha$ -sublevel sets are the half-plane areas defined by the level lines. In the middle image, the  $\alpha$ -sublevel sets are regions bounded by the coordinate axes and level sets.

In the bottom right image, with  $\alpha > 0$ , the level sets are the yellow or red lines. The corresponding  $\alpha$ -sublevel sets are the inward-contracted regions, limited by the red curves. These regions, as you can observe, are non-convex.

**Theorem:** If a function is convex, then all of its  $\alpha$ -sublevel sets are convex. The converse is not true: if all  $\alpha$ -sublevel sets of a function are convex, it does not necessarily mean the function is convex.

This indicates that if there exists an  $\alpha$ -sublevel set of a function that is nonconvex, the function itself is nonconvex. Hence, hyperbolic functions are not convex.

A function that is not convex but has convex  $\alpha$ -sublevel sets is  $f(x, y) = -e^{x+y}$ . This function has half-plane  $\alpha$ -sublevel sets, which are convex, but the function itself is not convex (in this case, it is concave).

Another example of a function with convex  $\alpha$ -sublevel sets but is not convex is shown below:

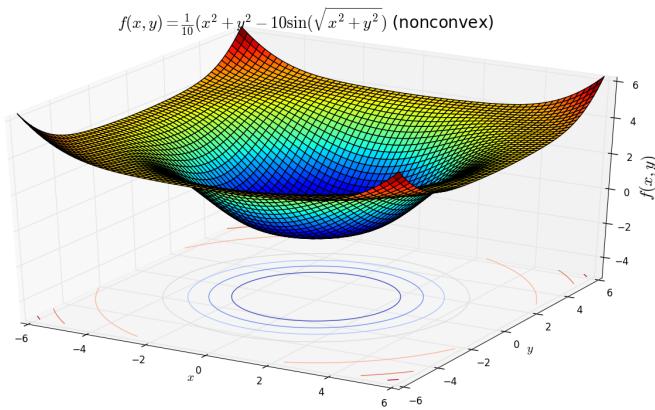


Figure 3.9: All alpha-sublevel sets are convex sets, but the function is non-convex.

All  $\alpha$ -sublevel sets of this function are circular, making them convex, but the function itself is not convex. One can find two points on the surface such that the straight line connecting them lies entirely below the surface (for example, a point on the "wing" and one in the "base").

### 3.3.5 Checking convexity using derivatives

There is a method to check if a differentiable function is convex using its first or second derivatives.

#### First-order condition

Let's first define the equation of the tangent line (or plane) of a differentiable function  $f$  at a point on its graph  $(\mathbf{x}_0, f(\mathbf{x}_0))$ . For a one-variable function, you are likely familiar with:

$$y = f'(x_0)(x - x_0) + f(x_0)$$

For multivariable functions, let  $\nabla f(\mathbf{x}_0)$  be the gradient of  $f$  at the point  $\mathbf{x}_0$ . The equation of the tangent plane is given by:

$$y = \nabla f(\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0) + f(\mathbf{x}_0)$$

**First-order condition** states: Suppose  $f$  is differentiable and defined on a convex set. Then,  $f$  is convex if and only if, for all  $\mathbf{x}, \mathbf{x}_0$  in the domain of  $f$ :

$$f(\mathbf{x}) \geq f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0) \quad (3.1)$$

Similarly,  $f$  is strictly convex if the equality in (3.1) holds if and only if  $\mathbf{x} = \mathbf{x}_0$ . Intuitively, a function is convex if the tangent line at any point on its graph lies below the graph. (Remember, the domain must be convex.)

Below are examples of a convex function and a non-convex function:

$f$  is differentiable with convex domain

$f$  is convex iff  $f(\mathbf{x}) \geq f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0), \forall \mathbf{x}, \mathbf{x}_0 \in \text{dom } f$

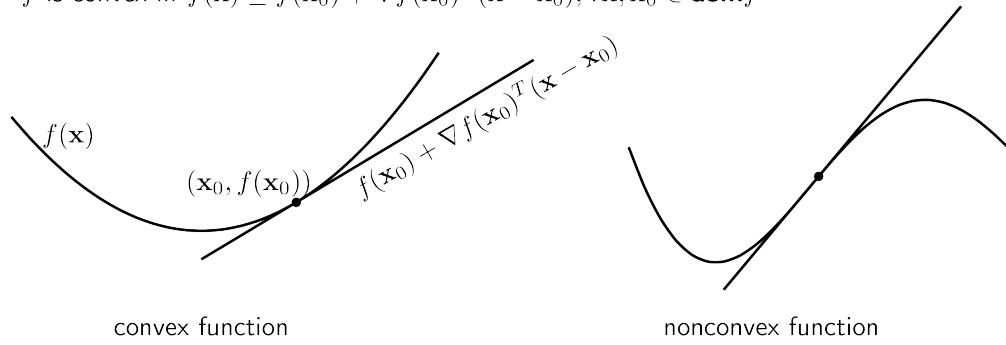


Figure 3.10: First-order convexity check. Left: convex function, right: non-convex function.

The function on the left is convex. The function on the right is non-convex because its graph is sometimes above and sometimes below its tangent line.

**Example:** If a symmetric matrix  $\mathbf{A}$  is positive definite, then the function  $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x}$  is convex.

*Proof:* The gradient of the function is:

$$\nabla f(\mathbf{x}) = 2\mathbf{A}\mathbf{x}$$

Thus, the first-order condition becomes (note that  $\mathbf{A}$  is symmetric):

$$\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 2(\mathbf{A} \mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) + \mathbf{x}_0^T \mathbf{A} \mathbf{x}_0 \quad \Rightarrow \quad (\mathbf{x} - \mathbf{x}_0)^T \mathbf{A} (\mathbf{x} - \mathbf{x}_0) \geq 0$$

The last inequality holds because  $\mathbf{A}$  is positive definite. Therefore,  $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x}$  is convex.

### Second-order condition

For multivariable functions with  $d$ -dimensional input, the first derivative is a vector of dimension  $d$ , and the second derivative is a  $d \times d$  matrix, known as the Hessian, denoted  $\nabla^2 f(\mathbf{x})$ .

**Second-order condition** states: A twice differentiable function  $f$  is convex if and only if its domain is convex and its Hessian is positive semi-definite at every point in the domain:

$$\nabla^2 f(\mathbf{x}) \succeq 0$$

If the Hessian is positive definite, then  $f$  is strictly convex.

#### Examples:

- The function  $f(x) = x \log(x)$  is strictly convex because its domain is  $x > 0$  (a convex set) and  $f''(x) = \frac{1}{x}$  is positive for all  $x$  in the domain.
- The function  $f(x) = x^2 + 5 \sin(x)$  is not convex because its second derivative,  $f''(x) = 2 - 5 \sin(x)$ , can be negative.
- The cross-entropy function is strictly convex. Consider two probabilities  $x$  and  $1 - x$ , and a constant  $a \in [0, 1]$ . The second derivative of  $f(x) = -(a \log(x) + (1-a) \log(1-x))$  is positive.
- If  $\mathbf{A}$  is positive definite, then  $f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x}$  is convex because its Hessian is  $\mathbf{A}$ .

### 3.3.6 Conclusion

Understanding convexity is essential in optimization problems, particularly because convex functions exhibit a unique global minimum, a highly desirable property when solving optimization tasks. Affine and quadratic functions are fundamental examples of convexity, while non-convex functions illustrate challenges in finding global optima. Norms, essential in various applications, further exemplify the broad relevance of convexity.

## 3.4 Matrix Diagonalization and Eigen Decomposition

You may remember a common problem in Linear Algebra: Matrix Diagonalization. This problem states that a square matrix  $\mathbf{A} \in \mathbb{R}^{n \times n}$  is called *diagonalizable* if there exists a diagonal matrix  $\mathbf{D}$  and an invertible matrix  $\mathbf{P}$  such that:

$$\mathbf{A} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1} \quad (1)$$

The number of non-zero elements in the diagonal matrix  $\mathbf{D}$  represents the rank of matrix  $\mathbf{A}$ .

Multiplying both sides of (1) by  $\mathbf{P}$  gives:

$$\mathbf{AP} = \mathbf{PD} \quad (2)$$

Let  $\mathbf{p}_i, \mathbf{d}_i$  denote the  $i$ -th column of matrices  $\mathbf{P}$  and  $\mathbf{D}$ , respectively. Since each column on both sides of (2) must be equal, we have:

$$\mathbf{Ap}_i = \mathbf{Pd}_i = d_{ii}\mathbf{p}_i \quad (3)$$

where  $d_{ii}$  is the  $i$ -th element of  $\mathbf{D}$ .

The second equality arises because  $\mathbf{D}$  is a diagonal matrix, meaning  $\mathbf{d}_i$  only has  $d_{ii}$  as a non-zero component. This expression (3) shows that each  $d_{ii}$  is an *eigenvalue* of  $\mathbf{A}$ , and each column vector  $\mathbf{p}_i$  is an *eigenvector* of  $\mathbf{A}$  corresponding to the eigenvalue  $d_{ii}$ .

The factorization of a square matrix as in (1) is known as *Eigen Decomposition*.

A key point is that the decomposition (1) only applies to square matrices and may not always exist. It only exists if  $\mathbf{A}$  has  $n$  linearly independent eigenvectors; otherwise, an invertible  $\mathbf{P}$  does not exist. Additionally, this decomposition is not unique, as if  $\mathbf{P}, \mathbf{D}$  satisfy (1), then  $k\mathbf{P}, \mathbf{D}$  also satisfy it for any non-zero real  $k$ .

Decomposing a matrix into products of special matrices (Matrix Factorization or Matrix Decomposition) has significant benefits, such as dimensionality reduction, data compression, exploring data characteristics, solving linear equations, clustering, and many other applications. Recommendation Systems are one of the many applications of Matrix Factorization.

In this article, I will introduce a beautiful Matrix Factorization method in Linear Algebra: Singular Value Decomposition (SVD). You will see that any matrix, not necessarily square, can be decomposed into the product of three special matrices.

### 3.4.1 Eigenvalues and Eigenvectors

For a square matrix  $\mathbf{A} \in \mathbb{R}^{n \times n}$ , a scalar  $\lambda$  and a non-zero vector  $\mathbf{x} \in \mathbb{R}^n$  are an eigenvalue and eigenvector, respectively, if:

$$\mathbf{Ax} = \lambda\mathbf{x}$$

### A few properties:

1. If  $\mathbf{x}$  is an eigenvector of  $\mathbf{A}$  associated with  $\lambda$ , then  $k\mathbf{x}, k \neq 0$  is also an eigenvector associated with  $\lambda$ .
2. Any square matrix of order  $n$  has  $n$  eigenvalues (including repetitions), which may be complex.
3. For symmetric matrices, all eigenvalues are real.
4. For a positive definite matrix, all its eigenvalues are positive real numbers. For a positive semi-definite matrix, all its eigenvalues are non-negative real numbers.

The last property can be derived from the definition of a (semi-)positive definite matrix. Indeed, let  $\mathbf{u} \neq \mathbf{0}$  be an eigenvector of a positive semi-definite matrix  $\mathbf{A}$  with eigenvalue  $\lambda$ , then:

$$\mathbf{A}\mathbf{u} = \lambda\mathbf{u} \Rightarrow \mathbf{u}^T\mathbf{A}\mathbf{u} = \lambda\mathbf{u}^T\mathbf{u} = \lambda\|\mathbf{u}\|_2^2$$

Since  $\mathbf{A}$  is positive semi-definite,  $\mathbf{u}^T\mathbf{A}\mathbf{u} \geq 0$  for all  $\mathbf{u} \neq \mathbf{0}$ , implying that  $\lambda$  is non-negative.

### 3.4.2 Orthogonal and Orthonormal Bases

A basis  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m \in \mathbb{R}^m\}$  is called *orthogonal* if each vector is non-zero and any two different vectors are orthogonal:

$$\mathbf{u}_i \neq \mathbf{0}; \quad \mathbf{u}_i^T \mathbf{u}_j = 0 \quad \forall 1 \leq i \neq j \leq m$$

An *orthonormal* basis  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m \in \mathbb{R}^m\}$  is an orthogonal basis in which each vector has a Euclidean length (2-norm) of 1:

$$\mathbf{u}_i^T \mathbf{u}_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \quad (3.2)$$

Let  $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m]$ , where  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m \in \mathbb{R}^m\}$  is orthonormal. Then, from (4), we have:

$$\mathbf{U}\mathbf{U}^T = \mathbf{U}^T\mathbf{U} = \mathbf{I}$$

where  $\mathbf{I}$  is the  $m$ -order identity matrix. We call  $\mathbf{U}$  an *orthogonal matrix*.

### Some properties:

1.  $\mathbf{U}^{-1} = \mathbf{U}^T$ : the inverse of an orthogonal matrix is its transpose.
2. If  $\mathbf{U}$  is orthogonal, then  $\mathbf{U}^T$  is also orthogonal.
3. The determinant of an orthogonal matrix is 1 or  $-1$ .
4. An orthogonal matrix represents a *rotation* of a vector. If we rotate vectors  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$  with an orthogonal matrix  $\mathbf{U} \in \mathbb{R}^{m \times m}$ , the inner product of the rotated vectors remains unchanged:

$$(\mathbf{U}\mathbf{x})^T(\mathbf{U}\mathbf{y}) = \mathbf{x}^T\mathbf{U}^T\mathbf{U}\mathbf{y} = \mathbf{x}^T\mathbf{y}$$

5. Let  $\hat{\mathbf{U}} \in \mathbb{R}^{m \times r}$ ,  $r < m$  be a submatrix of  $\mathbf{U}$  formed by  $r$  columns of  $\mathbf{U}$ , then  $\hat{\mathbf{U}}^T\hat{\mathbf{U}} = \mathbf{I}_r$ .

### 3.4.3 Normalizing Eigenvectors

The last task is to normalize the eigenvectors so they form an orthonormal system. This can be done based on two points:

**First**, if  $\mathbf{A}$  is a symmetric matrix, and  $(\lambda_1, \mathbf{x}_1), (\lambda_2, \mathbf{x}_2)$  are eigenvalue-eigenvector pairs of  $\mathbf{A}$  with  $\lambda_1 \neq \lambda_2$ , then  $\mathbf{x}_1^T \mathbf{x}_2 = 0$ . That is, any two vectors in different eigenspaces of a symmetric matrix are orthogonal. The proof is shown as follows:

$$\mathbf{x}_2^T \mathbf{A} \mathbf{x}_1 = \mathbf{x}_1^T \mathbf{A} \mathbf{x}_2 = \lambda_1 \mathbf{x}_2^T \mathbf{x}_1 = \lambda_2 \mathbf{x}_1^T \mathbf{x}_2 \Rightarrow \mathbf{x}_1^T \mathbf{x}_2 = 0 \quad (3.3)$$

**Second**, for independent eigenvectors in the same eigenspace, we can use the Gram-Schmidt process to normalize them to form an orthonormal system.

Combining these points, we obtain eigenvectors that form an orthonormal system, which is the matrix  $\mathbf{U}_K$  in PCA.

## 3.5 Singular Value Decomposition (SVD)

To clarify the dimensions of each matrix, we denote matrices with their dimensions, e.g.,  $\mathbf{A}_{m \times n}$  represents  $\mathbf{A} \in \mathbb{R}^{m \times n}$ .

### 3.5.1 Definition of SVD

Any matrix  $\mathbf{A}_{m \times n}$  can be decomposed as:

$$\mathbf{A}_{m \times n} = \mathbf{U}_{m \times m} \Sigma_{m \times n} (\mathbf{V}_{n \times n})^T \quad (3.4)$$

where  $\mathbf{U}, \mathbf{V}$  are *orthogonal matrices*, and  $\Sigma$  is a *diagonal, non-square matrix* with diagonal elements  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r \geq 0$  where  $r$  is the rank of  $\mathbf{A}$ . Although  $\Sigma$  is not square, we can still consider it diagonal if its non-zero elements only appear on the diagonal.

The number of non-zero elements in  $\Sigma$  represents the rank of  $\mathbf{A}$ .

If you want to see the proof of the existence of SVD, you can find it [here<sup>2</sup>](#).

Note that the representation of 3.4 is not unique, as we can change the signs of both  $\mathbf{U}$  and  $\mathbf{V}$  without affecting 3.4. Nevertheless, people still refer to it as *the SVD*.

Figure 3.11 shows the SVD of matrix  $\mathbf{A}_{m \times n}$  in two cases:  $m < n$  and  $m > n$ . The case  $m = n$  fits into either scenario.

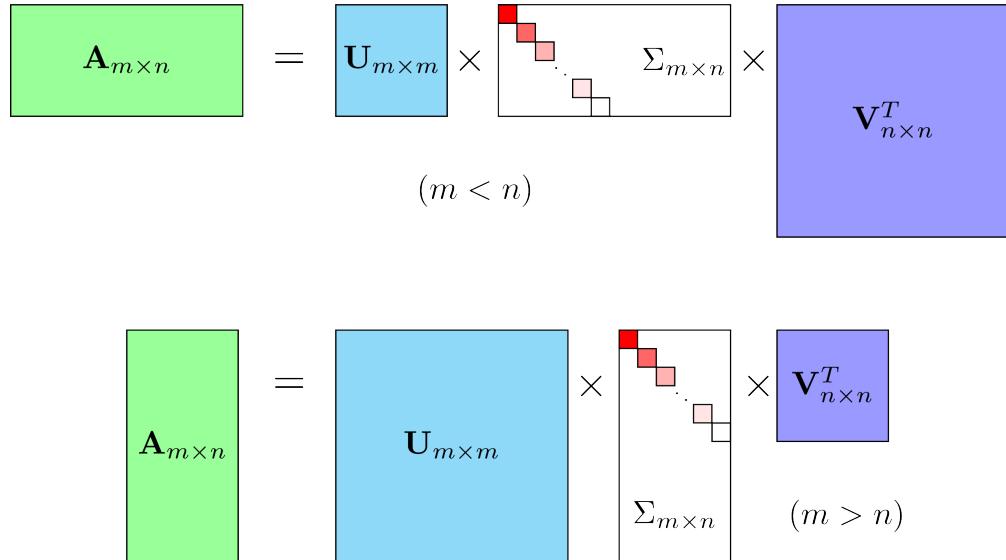


Figure 3.11: SVD of matrix  $\mathbf{A}$  when:  $m < n$  (top), and  $m > n$  (bottom).  $\Sigma$  is a diagonal matrix with non-negative, descending elements. Darker red indicates larger values. White cells represent zeros in the matrix.

<sup>2</sup><http://db.cs.duke.edu/courses/cps111/spring07/notes/12.pdf>

### 3.5.2 Origin of the Term Singular Value Decomposition

Ignoring the dimensions of each matrix, from 3.4, we can conclude:

$$\begin{aligned}\mathbf{AA}^T &= \mathbf{U}\Sigma\mathbf{V}^T(\mathbf{U}\Sigma\mathbf{V}^T)^T \\ &= \mathbf{U}\Sigma\mathbf{V}^T\mathbf{V}\Sigma^T\mathbf{U}^T \\ &= \mathbf{U}\Sigma\Sigma^T\mathbf{U}^T = \mathbf{U}\Sigma\Sigma^T\mathbf{U}^{-1}\end{aligned}\tag{3.5}$$

The last equality arises because  $\mathbf{V}^T\mathbf{V} = \mathbf{I}$  since  $\mathbf{V}$  is orthogonal.

Observing that  $\Sigma\Sigma^T$  is a diagonal matrix with diagonal elements  $\sigma_1^2, \sigma_2^2, \dots$ , equation 3.5 represents the Eigen Decomposition of  $\mathbf{AA}^T$ . Furthermore,  $\sigma_1^2, \sigma_2^2, \dots$  are the eigenvalues of  $\mathbf{AA}^T$ .

Since  $\mathbf{AA}^T$  is positive semi-definite, its eigenvalues are non-negative. The  $\sigma_i$  values, which are the square roots of the eigenvalues of  $\mathbf{AA}^T$ , are called the *singular values* of  $\mathbf{A}$ . This is the origin of the term Singular Value Decomposition.

Each column of  $\mathbf{U}$  is an eigenvector of  $\mathbf{AA}^T$ , called a *left-singular vector* of  $\mathbf{A}$ . Similarly,  $\mathbf{A}^T\mathbf{A} = \mathbf{V}\Sigma^T\Sigma\mathbf{V}^T$ , and the columns of  $\mathbf{V}$  are called the *right-singular vectors* of  $\mathbf{A}$ .

In Python, to compute the SVD of a matrix, use the ‘linalg‘ module in ‘numpy‘:

Listing 3.1: SVD Computation

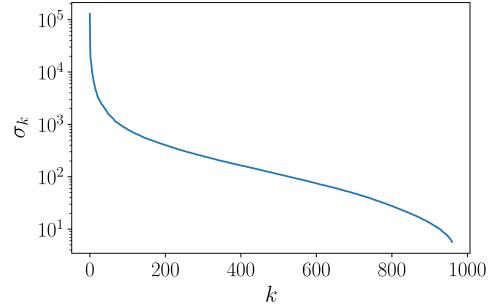
```

1 import numpy as np
2 from numpy import linalg as LA
3
4 m, n = 2, 3
5 A = np.random.rand(m, n)
6
7 U, S, V = LA.svd(A)
8
9 # Checking if U, V are orthogonal and S is a diagonal matrix with
10 # nonnegative decreasing elements
11 print "Frobenius norm of (UU^T - I) ="
12 LA.norm(U.dot(U.T) - np.eye(m))
13 print "\n S = ", S, "\n"
14 print "Frobenius norm of (VV^T - I) ="
15 LA.norm(V.dot(V.T) - np.eye(n))
```

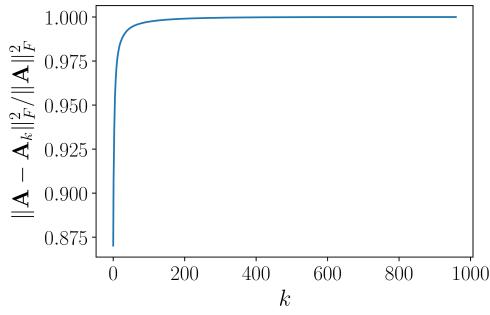
### 3.5.3 SVD in Image Compression



(a) The original grayscale image, represented as a matrix of size  $960 \times 1440$ .



(b) The values of the singular values of the image matrix in log scale. A rapid decrease is observed around  $k = 200$ .



(c) The amount of information retained for different values of  $k$ . From  $k = 200$  onwards, nearly all the information is retained. Therefore, the image matrix can be approximated with a lower-rank matrix.

Figure 3.12: Example of SVD for images.

To store an image using Truncated SVD, we will save the matrices  $\mathbf{U}_k \in \mathbb{R}^{m \times k}, \Sigma_k \in \mathbb{R}^{k \times k}, \mathbf{V}_k \in \mathbb{R}^{n \times k}$ . The total number of elements to store is  $k(m + n + 1)$ , noting that only the diagonal values of  $\Sigma_k$  need to be stored. Assuming each element is stored as a 4-byte real number, the required storage is  $4k(m + n + 1)$  bytes. Comparing this with the original image of size  $mn$ , where each value is stored as a 1-byte integer, the compression ratio is:

$$\frac{4k(m + n + 1)}{mn}$$

When  $k \ll m, n$ , this ratio is less than 1. In our example with  $m = 960, n = 1440, k = 100$ , the compression ratio is approximately 0.69, indicating a 30% reduction in storage.

### 3.5.4 Conclusion

- Besides the two applications mentioned above, SVD is also closely related to the Moore-Penrose pseudoinverse. The pseudoinverse plays a significant role in solving linear equations.
- SVD has many other interesting properties and applications, which we will explore gradually. First is its role in Dimensionality Reduction with Principal Component Analysis, which will be discussed in the next chapter.
- When the matrix  $\mathbf{A}$  is large, computing its SVD can be time-consuming. Calculating Truncated SVD with a small  $k$  by using full SVD is impractical. Instead, there is an iterative method to efficiently compute the largest  $k$  eigenvalues and eigenvectors of a large matrix  $\mathbf{AA}^T$ , which saves significant time. Readers can refer to Power method for approximating eigenvalues.

## 3.6 Principal Component Analysis (PCA)

### 3.6.1 Introduction

Dimensionality Reduction is one of the key techniques in Machine Learning. Feature vectors in practical problems can have very high dimensions, up to several thousand. Additionally, the number of data points is also typically large. Direct storage and computation on such high-dimensional data can pose difficulties regarding both storage and processing speed. Therefore, dimensionality reduction is a crucial step in many problems and is considered a form of data compression.

Dimensionality Reduction, simply put, is the process of finding a function, where this function takes as input an initial data point  $\mathbf{x} \in \mathbb{R}^D$  with  $D$  being very large, and creates a new data point  $\mathbf{z} \in \mathbb{R}^K$  with  $K < D$ .

As usual, I will present the simplest method among Dimensionality Reduction algorithms based on a linear model. This method is called *Principal Component Analysis* (PCA). This method is based on the observation that data is often not randomly distributed in space but usually distributed near certain special curves or surfaces. PCA examines a particular case when these surfaces are linear, called subspaces.

To transform a vector of  $D$  dimensions into a vector of  $K$  dimensions, the simplest way is to multiply a matrix of  $K \times D$  with that vector, as shown in Figure 3.13 below.

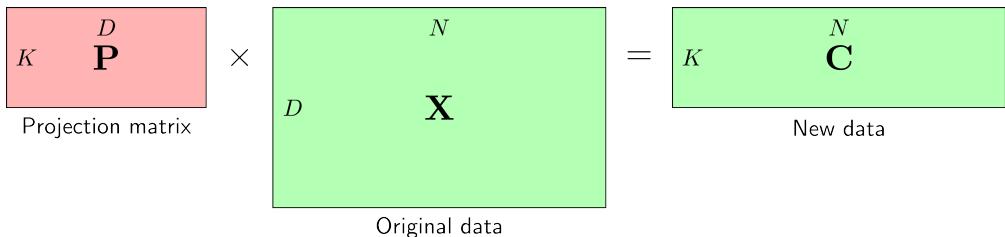


Figure 3.13: Matrix Projection for Dimensionality Reduction. The original data is the columns of the matrix  $\mathbf{X}$ . The matrix  $\mathbf{P} \in \mathbb{R}^{K \times D}$  projects each column of the data matrix onto a new lower-dimensional space. The new data is given in the matrix  $\mathbf{C}$ .

The matrix  $\mathbf{P} \in \mathbb{R}^{K \times D}$  serves to *project* each column  $\mathbf{x}$  of the original data matrix onto a lower-dimensional space. In the new space, each data point  $\mathbf{x}$  is represented by a column  $\mathbf{c}$  in the new data matrix  $\mathbf{C}$ .

Two questions arise: what should the structure of matrix  $\mathbf{P}$  be to minimize information loss, and how should the dimension  $K$  of the new data be chosen? PCA will help us answer these questions.

Before delving into the details of PCA, let us review a bit of Linear Algebra and Statistics.

### 3.6.2 Matrix 2-Norm

We often mention vector norms, but we have not worked much with matrix norms (aside from the Frobenius norm). In this section, we will introduce a class of matrix norms defined based on vector norms, also known as *Induced Norms*.

Let  $\|\mathbf{x}\|_\alpha$  be any norm of the vector  $\mathbf{x}$ . Based on this norm, we define the corresponding norm for a matrix  $\mathbf{A}$ :

$$\|\mathbf{A}\|_\alpha = \max_{\mathbf{x}} \frac{\|\mathbf{Ax}\|_\alpha}{\|\mathbf{x}\|_\alpha}$$

Note that matrix  $\mathbf{A}$  does not need to be square, and its number of columns matches the dimension of  $\mathbf{x}$ . Calculating the matrix norm becomes an optimization problem.

We are particularly interested in the 2-norm. The 2-norm of a matrix is defined as:

$$\|\mathbf{A}\|_2 = \max_{\mathbf{x}} \frac{\|\mathbf{Ax}\|_2}{\|\mathbf{x}\|_2} \quad (3.6)$$

If  $\mathbf{x}$  is the solution to the optimization problem (1), then  $k\mathbf{x}$  is also a solution for any non-zero scalar  $k$ . Without loss of generality, we assume the denominator equals 1. The optimization problem (1) can then be rewritten as:

$$\|\mathbf{A}\|_2 = \max_{\|\mathbf{x}\|_2=1} \|\mathbf{Ax}\|_2 \quad (3.7)$$

This translates to finding  $\mathbf{x}$  such that:

$$\mathbf{x} = \operatorname{argmax}_{\mathbf{x}} \|\mathbf{Ax}\|_2^2 \quad (3.8)$$

$$\text{s.t.:} \quad \|\mathbf{x}\|_2^2 = 1 \quad (3.9)$$

Here, we square the 2-norms to avoid square roots. Problem 3.9 can be solved using the *Lagrange Multiplier Method* due to the constraint equation.

The Lagrangian of problem 3.9 is:

$$\mathcal{L}(\mathbf{x}, \lambda) = \|\mathbf{Ax}\|_2^2 + \lambda(1 - \|\mathbf{x}\|_2^2)$$

The solution of problem 3.9 will satisfy:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = 2\mathbf{A}^T \mathbf{Ax} - 2\lambda \mathbf{x} = \mathbf{0} \quad (3.10)$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = 1 - \|\mathbf{x}\|_2^2 = 0 \quad (3.11)$$

From 3.10, we have:

$$\mathbf{A}^T \mathbf{Ax} = \lambda \mathbf{x} \quad (3.12)$$

This implies  $\lambda$  is an eigenvalue of  $\mathbf{A}^T \mathbf{A}$ , and  $\mathbf{x}$  is the corresponding eigenvector. By multiplying both sides of 3.12 by  $\mathbf{x}^T$  from the left, we get:

$$\mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} = \lambda \mathbf{x}^T \mathbf{x} = \lambda$$

The left side is  $\|\mathbf{A}\mathbf{x}\|_2^2$ , which is the objective function in 3.9. The objective reaches its maximum when  $\lambda$  reaches its maximum value. Thus,  $\lambda$  is the largest eigenvalue of  $\mathbf{A}^T \mathbf{A}$  or, equivalently, the largest singular value of  $\mathbf{A}$ .

Therefore, the 2-norm of a matrix is the largest singular value of that matrix. The solution of problem 3.9 is a *right-singular vector* associated with that singular value.

Using similar reasoning, we can conclude that the problem:

$$\min_{\|\mathbf{x}\|_2=1} \mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x}$$

has a solution that is the eigenvector corresponding to the smallest eigenvalue of  $\mathbf{A}^T \mathbf{A}$ . The minimum objective value equals this smallest eigenvalue.

### 3.6.3 Vector Representation in Different Bases

In  $D$ -dimensional space, the coordinates of each point are defined relative to a specific coordinate system. With different coordinate systems, the coordinates of each point will differ.

The set of vectors  $\mathbf{e}_1, \dots, \mathbf{e}_D$ , where each vector  $\mathbf{e}_d$  has exactly one non-zero element in the  $d$ -th component with that element equal to 1, is known as the standard basis in  $D$ -dimensional space. Arranging these vectors in order gives the identity matrix of dimension  $D$ .

Each column vector  $\mathbf{x} = [x_1, x_2, \dots, x_D] \in \mathbb{R}^D$  is represented in the standard basis as:

$$\mathbf{x} = x_1 \mathbf{e}_1 + x_2 \mathbf{e}_2 + \cdots + x_D \mathbf{e}_D$$

Suppose there is another basis  $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_D$  (these vectors are linearly independent), then the representation of  $\mathbf{x}$  in this new basis is:

$$\mathbf{x} = y_1 \mathbf{u}_1 + y_2 \mathbf{u}_2 + \cdots + y_D \mathbf{u}_D = \mathbf{Uy}$$

Here,  $\mathbf{U}$  is the matrix whose  $d$ -th column is  $\mathbf{u}_d$ . The vector  $\mathbf{y}$  represents  $\mathbf{x}$  in the new basis, where  $\mathbf{y}$  can be calculated by:

$$\mathbf{y} = \mathbf{U}^{-1} \mathbf{x} \tag{3.13}$$

Since  $\mathbf{U}$  is invertible (its columns are linearly independent), orthogonal matrices  $\mathbf{U}$  have a particular interest due to their properties. For orthogonal matrices, we have:

$$\mathbf{U}^{-1} = \mathbf{U}^T$$

Thus,  $\mathbf{y}$  in 3.13 can be computed as:

$$\mathbf{y} = \mathbf{U}^T \mathbf{x}$$

From this, we deduce:  $y_i = \mathbf{x}^T \mathbf{u}_i = \mathbf{u}_i^T \mathbf{x}, i = 1, 2, \dots, D.$

Note that the zero vector is represented the same way in all bases. Figure 3.14 below illustrates changing the basis:

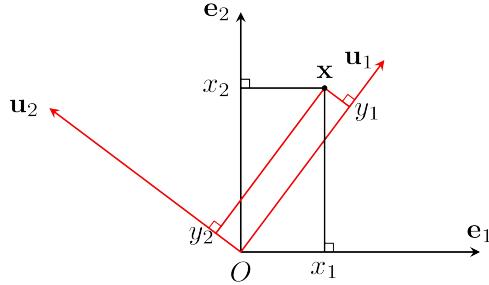


Figure 3.14: Coordinate transformation in different bases.

Coordinate transformations using an orthogonal matrix can be seen as a rotation of the coordinate axes, or conversely, as a rotation of the data vectors in the opposite direction.

### 3.6.4 Trace

The trace function, defined on the set of square matrices, is widely used in optimization due to its convenient properties. The trace returns the sum of the diagonal elements of a square matrix.

Key properties of the trace function are as follows:

- $\text{trace}(\mathbf{A}) = \text{trace}(\mathbf{A}^T)$
- $\text{trace}(k\mathbf{A}) = k\text{trace}(\mathbf{A})$  with  $k$  as a scalar.
- $\text{trace}(\mathbf{AB}) = \text{trace}(\mathbf{BA})$
- $\|\mathbf{A}\|_F^2 = \text{trace}(\mathbf{A}^T \mathbf{A}) = \text{trace}(\mathbf{AA}^T)$ , with  $\mathbf{A}$  being any matrix.
- $\text{trace}(\mathbf{A}) = \sum_{i=1}^D \lambda_i$ , where  $\mathbf{A}$  is square, and  $\lambda_i$  are its eigenvalues.

### 3.6.5 Expectation and Covariance Matrix

### 3.6.6 Univariate Data

Given  $N$  values  $x_1, x_2, \dots, x_N$ , the *expectation* and *variance* are defined as:

$$\bar{x} = \frac{1}{N} \sum_{n=1}^N x_n = \frac{1}{N} \mathbf{X} \mathbf{1} \quad (3.14)$$

$$\sigma^2 = \frac{1}{N} \sum_{n=1}^N (x_n - \bar{x})^2 \quad (3.15)$$

where  $\mathbf{1} \in \mathbb{R}^N$  is a vector with all elements equal to 1.

### 3.6.7 Multivariate Data

Given  $N$  data points represented by column vectors  $\mathbf{x}_1, \dots, \mathbf{x}_N$ , the *expectation vector* and *covariance matrix* are defined as:

$$\bar{\mathbf{x}} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \quad (3.16)$$

$$\mathbf{S} = \frac{1}{N} \sum_{n=1}^N (\mathbf{x}_n - \bar{\mathbf{x}})(\mathbf{x}_n - \bar{\mathbf{x}})^T = \frac{1}{N} \hat{\mathbf{X}} \hat{\mathbf{X}}^T \quad (3.17)$$

where  $\hat{\mathbf{X}}$  is obtained by subtracting  $\bar{\mathbf{x}}$  from each column of  $\mathbf{X}$ :

$$\hat{\mathbf{x}}_n = \mathbf{x}_n - \bar{\mathbf{x}}$$

Key points to note:

- The covariance matrix is symmetric and positive semi-definite.
- All diagonal elements are non-negative, representing the variance of each dimension.
- Off-diagonal elements  $s_{ij}, i \neq j$  represent the covariance between dimensions  $i$  and  $j$ , indicating correlation. If 0, dimensions  $i$  and  $j$  are uncorrelated.
- A diagonal covariance matrix indicates fully uncorrelated data.

### 3.6.8 Principal Component Analysis

The simplest way to reduce the dimensionality of data from  $D$  to  $K < D$  is to keep only the  $K$  most *important* components. However, this approach is likely suboptimal because

we do not yet know which components are more important. Or in the worst case, if each component carries the same amount of information, discarding any component could lead to significant information loss.

However, if we can represent the original data vectors in a new basis where the *importance* of each component differs significantly, then we can disregard the least important components.

Take, for example, two cameras capturing an image of a person: one camera positioned in front of the person and one above. Clearly, the image captured by the front-facing camera holds more information than the top-down view, so the overhead shot can be omitted without losing much detail about the person's shape.

PCA is a method for finding a new basis so that the primary information in the data is concentrated in a few coordinates, while the remaining coordinates carry minimal information. To simplify calculations, PCA seeks an orthonormal basis for the new basis.

Suppose the new orthonormal basis is  $\mathbf{U}$ , and we wish to retain  $K$  coordinates in this basis. Without loss of generality, let these be the first  $K$  components. See 3.15 below:

$$\begin{aligned}
 \begin{matrix} & N \\ D & \mathbf{X} \end{matrix} &= \begin{matrix} & K & D-K \\ D \mathbf{U}_K & \bar{\mathbf{U}}_K \end{matrix} \times \begin{matrix} & N \\ K & \mathbf{Z} \\ D-K & \mathbf{Y} \end{matrix} \\
 \text{Original data} & \quad \text{An orthogonal matrix} \quad \text{Coordinates in new basis} \\
 \\ 
 &= \begin{matrix} & K \\ D \mathbf{U}_K \end{matrix} \times \begin{matrix} & N \\ K & \mathbf{Z} \\ D \end{matrix} + \begin{matrix} & \mathbf{Y} \\ \bar{\mathbf{U}}_K \end{matrix} \times \begin{matrix} & \mathbf{Y} \end{matrix}
 \end{aligned}$$

Figure 3.15: The main idea of PCA: Find a new orthonormal basis such that the most important components are in the first  $K$  components.

In the new basis  $\mathbf{U} = [\mathbf{U}_K, \bar{\mathbf{U}}_K]$ , an orthonormal basis with  $\mathbf{U}_K$  being the submatrix formed by the first  $K$  columns of  $\mathbf{U}$ , the data matrix can be expressed as:

$$\mathbf{X} = \mathbf{U}_K \mathbf{Z} + \bar{\mathbf{U}}_K \mathbf{Y} \quad (3.18)$$

This leads to:

$$\begin{bmatrix} \mathbf{Z} \\ \mathbf{Y} \end{bmatrix} = \begin{bmatrix} \mathbf{U}_K^T \\ \bar{\mathbf{U}}_K^T \end{bmatrix} \mathbf{X} \Rightarrow \begin{cases} \mathbf{Z} = \mathbf{U}_K^T \mathbf{X} \\ \mathbf{Y} = \bar{\mathbf{U}}_K^T \mathbf{X} \end{cases} \quad (3.19)$$

The goal of PCA is to find the orthogonal matrix  $\mathbf{U}$  such that most information is retained in the green section  $\mathbf{U}_K \mathbf{Z}$ , while the red section  $\bar{\mathbf{U}}_K \mathbf{Y}$  can be approximated by a matrix independent of individual data points. That is, we will approximate  $\mathbf{Y}$  with a matrix where all columns are the same. Let each column be represented by  $\mathbf{b}$  (bias); thus, we approximate:

$$\mathbf{Y} \approx \mathbf{b} \mathbf{1}^T$$

where  $\mathbf{1}^T \in \mathbb{R}^{1 \times N}$  is a row vector with all elements equal to 1. Assuming  $\mathbf{U}$  has been found, we then solve for  $\mathbf{b}$ :

$$\mathbf{b} = \operatorname{argmin}_{\mathbf{b}} \| \| \mathbf{Y} - \mathbf{b} \mathbf{1}^T \| \|_F^2 = \operatorname{argmin}_{\mathbf{b}} \| \| \bar{\mathbf{U}}_K^T \mathbf{X} - \mathbf{b} \mathbf{1}^T \| \|_F^2$$

Solving for  $\mathbf{b}$  by setting the derivative of the objective function to zero gives:

$$(\mathbf{b} \mathbf{1}^T - \bar{\mathbf{U}}_K^T \mathbf{X}) \mathbf{1} = 0 \Rightarrow N \mathbf{b} = \bar{\mathbf{U}}_K^T \mathbf{X} \mathbf{1} \Rightarrow \mathbf{b} = \bar{\mathbf{U}}_K^T \bar{\mathbf{x}}$$

Thus, **computations become much simpler if the expectation vector  $\bar{\mathbf{x}} = \mathbf{0}$** . This can be achieved by initially subtracting the mean vector of all data points from each data vector, the first step in PCA.

Using this value of  $\mathbf{b}$ , the original data can be approximated as:

$$\mathbf{X} \approx \tilde{\mathbf{X}} = \mathbf{U}_K \mathbf{Z} + \bar{\mathbf{U}}_K \bar{\mathbf{U}}_K^T \bar{\mathbf{x}} \mathbf{1}^T \quad (3.20)$$

Combining 3.18 and 3.20, we define the loss function as follows:

$$J = \frac{1}{N} \| \| \mathbf{X} - \tilde{\mathbf{X}} \| \|_F^2 = \frac{1}{N} \| \| \bar{\mathbf{U}}_K \bar{\mathbf{U}}_K^T \mathbf{X} - \bar{\mathbf{U}}_K \bar{\mathbf{U}}_K^T \bar{\mathbf{x}} \mathbf{1}^T \| \|_F^2 \quad (3.21)$$

If the columns of a matrix  $\mathbf{V}$  form an orthonormal set, then for any matrix  $\mathbf{W}$ , we have:

$$\| \| \mathbf{V} \mathbf{W} \| \|_F^2 = \operatorname{trace}(\mathbf{W}^T \mathbf{V}^T \mathbf{V} \mathbf{W}) = \operatorname{trace}(\mathbf{W}^T \mathbf{W}) = \| \| \mathbf{W} \| \|_F^2$$

Thus, the loss function in 3.21 can be rewritten as:

$$J = \frac{1}{N} \| \| \bar{\mathbf{U}}_K^T (\mathbf{X} - \bar{\mathbf{x}} \mathbf{1}^T)^T \| \|_F^2 = \frac{1}{N} \| \| \bar{\mathbf{U}}_K^T \hat{\mathbf{X}} \| \|_F^2 \quad (3.22)$$

$$= \frac{1}{N} \| \| \hat{\mathbf{X}}^T \bar{\mathbf{U}}_K \| \|_F^2 = \frac{1}{N} \sum_{i=K+1}^D \| \| \hat{\mathbf{X}}^T \mathbf{u}_i \| \|_2^2 \quad (3.23)$$

$$= \frac{1}{N} \sum_{i=K+1}^D \mathbf{u}_i^T \hat{\mathbf{X}} \hat{\mathbf{X}}^T \mathbf{u}_i \quad (3.24)$$

$$= \sum_{i=K+1}^D \mathbf{u}_i^T \mathbf{S} \mathbf{u}_i \quad (3.25)$$

With  $\hat{\mathbf{X}} = \mathbf{X} - \bar{\mathbf{x}} \mathbf{1}^T$  as the normalized data and  $\mathbf{S}$  as the data covariance matrix. This matrix  $\hat{\mathbf{X}}$  is called the *zero-centered data*.

The remaining task is to find the  $\mathbf{u}_i$  to minimize the loss. Let  $\mathbf{S} = \frac{1}{N} \hat{\mathbf{X}}^T \hat{\mathbf{X}}$ . For any orthogonal matrix  $\mathbf{U}$ , substituting  $K = 0$  into 3.25 yields:

$$L = \sum_{i=1}^D \mathbf{u}_i^T \mathbf{S} \mathbf{u}_i = \frac{1}{N} \| \| \hat{\mathbf{X}}^T \mathbf{U} \| \|_F^2$$

Given  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_D \geq 0$ , eigenvalues of the positive semi-definite matrix  $\mathbf{S}$ , we conclude:

**Theorem 1:**  $F$  achieves its maximum value  $\sum_{i=1}^K \lambda_i$  when  $\mathbf{u}_i$  are eigenvectors with 2-norms of 1 associated with these eigenvalues. Of course, we do not forget the orthogonality condition among the  $\mathbf{u}_i$ .

The eigenvalues  $\lambda_i, i = 1, \dots, K$  represent the top  $K$  principal components. This method is thus known as *Principal Component Analysis*. To gain an intuitive view, see the figure below:

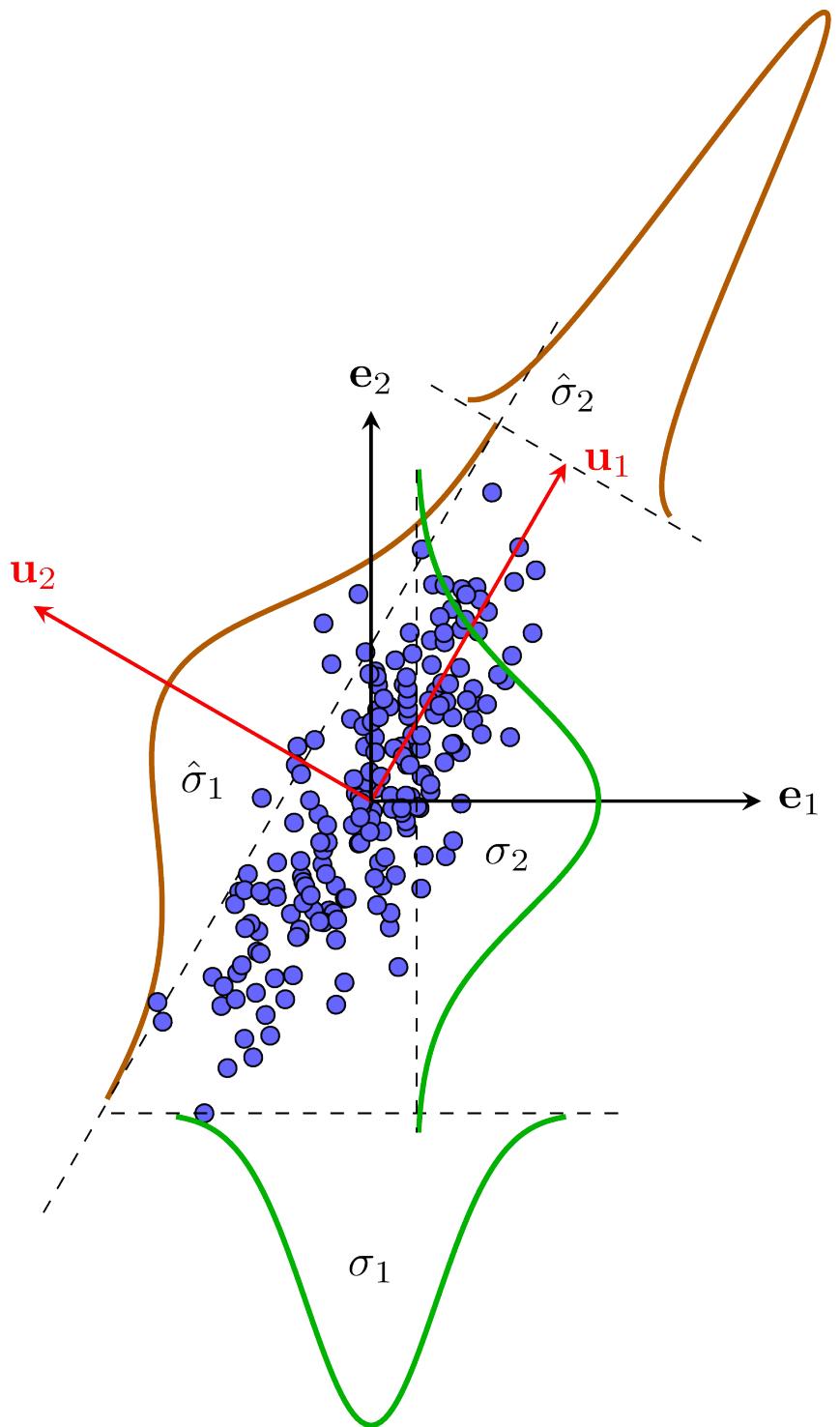


Figure 3.16: PCA from a statistical perspective. PCA finds a new orthonormal basis that acts as a rotation, so in this basis, certain dimensions have very low variance and can be ignored.

In the original space, the black basis vectors  $\mathbf{e}_1, \mathbf{e}_2$  have high variance along each data dimension.

sion. In the new space with red basis vectors  $\mathbf{u}_1, \mathbf{u}_2$ , the variance in the second dimension  $\hat{\sigma}_2$  is much smaller than  $\hat{\sigma}_1$ . Therefore, when projecting data onto  $\mathbf{u}_2$ , the points are close together. In this case, we can substitute coordinates in the  $\mathbf{u}_2$  direction by zero.

Thus, PCA finds a rotation with an orthogonal matrix, making it a method for dimensionality reduction while retaining the highest possible total variance.

The steps of PCA are summarized as follows:

1. Calculate the mean vector for all data:

$$\bar{\mathbf{x}} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n$$

2. Subtract the mean vector from each data point:

$$\hat{\mathbf{x}}_n = \mathbf{x}_n - \bar{\mathbf{x}}$$

3. Calculate the covariance matrix:

$$\mathbf{S} = \frac{1}{N} \hat{\mathbf{X}} \hat{\mathbf{X}}^T$$

4. Compute the eigenvalues and eigenvectors (norm 1) of this matrix and arrange them in descending order.
5. Choose the top  $K$  eigenvectors to construct the matrix  $\mathbf{U}_K$ , forming an orthogonal basis.
6. Project the normalized data  $\hat{\mathbf{X}}$  onto the new subspace.
7. The new data is given by:

$$\mathbf{Z} = \mathbf{U}_K^T \hat{\mathbf{X}}$$

The original data can be approximated as:

$$\mathbf{x} \approx \mathbf{U}_K \mathbf{Z} + \bar{\mathbf{x}}$$

## PCA procedure

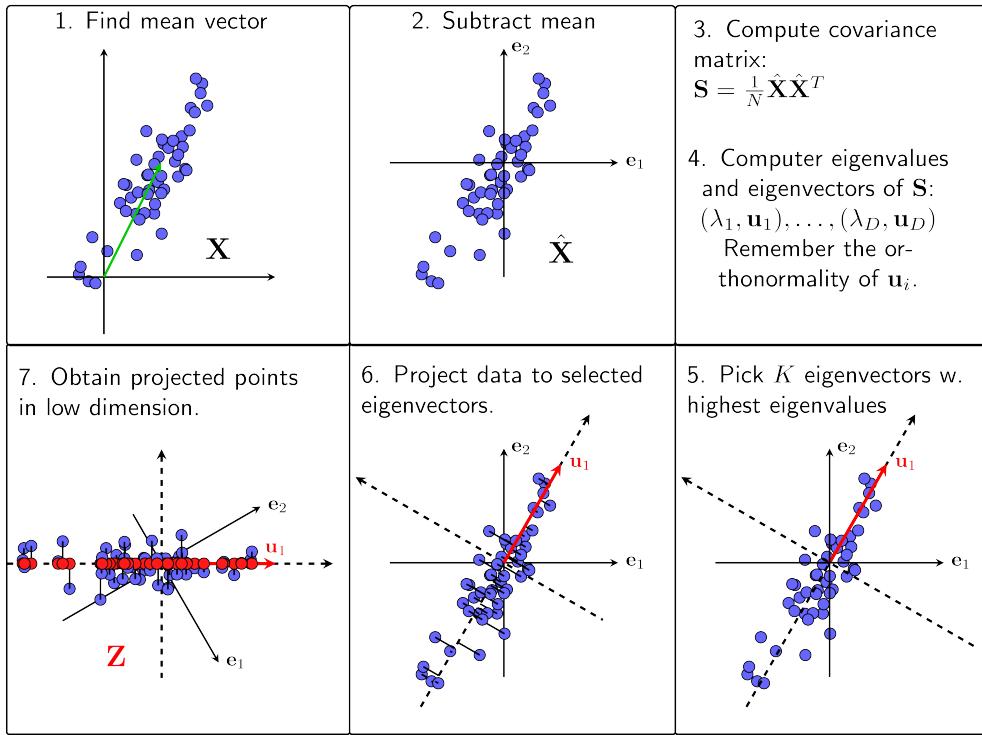


Figure 3.17: Steps of PCA

### 3.6.9 Relationship between PCA and SVD

There is a special relationship between PCA and SVD. To recognize this, let me recall two points discussed earlier:

### 3.6.10 SVD for the Best Low-Rank Approximation Problem

The solution  $\mathbf{A}$  of the problem of approximating a matrix by another matrix with a rank not exceeding  $k$ :

$$\min_{\mathbf{A}} \quad \| \|\mathbf{X} - \mathbf{A} \| \|_F \quad (3.26)$$

$$\text{s.t.} \quad \text{rank}(\mathbf{A}) = K \quad (3.27)$$

is the Truncated SVD of  $\mathbf{A}$ . Specifically, if the SVD of  $\mathbf{X} \in \mathbb{R}^{D \times N}$  is:

$$\mathbf{X} = \mathbf{U} \Sigma \mathbf{V}^T$$

where  $\mathbf{U} \in \mathbb{R}^{D \times D}$  and  $\mathbf{V} \in \mathbb{R}^{N \times N}$  are orthogonal matrices, and  $\Sigma \in \mathbb{R}^{D \times N}$  is a diagonal matrix (not necessarily square) with non-negative, descending elements on the diagonal, then the solution to problem 3.27 is:

$$\mathbf{A} = \mathbf{U}_K \Sigma_K \mathbf{V}_K^T \quad (3.28)$$

where  $\mathbf{U}_K \in \mathbb{R}^{D \times K}$  and  $\mathbf{V}_K \in \mathbb{R}^{N \times K}$  are matrices formed by the first  $K$  columns of  $\mathbf{U}$  and  $\mathbf{V}$ , and  $\Sigma_K \in \mathbb{R}^{K \times K}$  is a diagonal submatrix corresponding to the first  $K$  rows and columns of  $\Sigma$ .

### 3.6.11 The Idea of PCA

PCA aims to find an orthogonal matrix  $\mathbf{U}$  and a low-dimensional data representation  $\mathbf{Z}$  such that the following approximation is optimal:

$$\mathbf{X} \approx \tilde{\mathbf{X}} = \mathbf{U}_K \mathbf{Z} + \bar{\mathbf{U}}_K \bar{\mathbf{U}}_K^T \bar{\mathbf{x}} \mathbf{1}^T \quad (3.29)$$

where  $\mathbf{U}_K$  and  $\bar{\mathbf{U}}_K$  are matrices formed by the first  $K$  columns and the last  $D - K$  columns of the orthogonal matrix  $\mathbf{U}$ , and  $\bar{\mathbf{x}}$  is the mean vector of the data.

**Assuming the mean vector  $\bar{\mathbf{x}} = \mathbf{0}$ ,** then 3.29 simplifies to:

$$\mathbf{X} \approx \tilde{\mathbf{X}} = \mathbf{U}_K \mathbf{Z} \quad (3.30)$$

The PCA optimization problem becomes:

$$\mathbf{U}_K, \mathbf{Z} = \min_{\mathbf{U}_K, \mathbf{Z}} \| \| \mathbf{X} - \mathbf{U}_K \mathbf{Z} \| \|_F \quad (3.31)$$

$$\text{s.t.:} \quad \mathbf{U}_K^T \mathbf{U}_K = \mathbf{I}_K \quad (3.32)$$

where  $\mathbf{I}_K \in \mathbb{R}^{K \times K}$  is the identity matrix in  $K$ -dimensional space, ensuring that the columns of  $\mathbf{U}_K$  form an orthonormal system.

### 3.6.12 Relationship between PCA and SVD

Do you recognize the similarity between optimization problems 3.27 and 3.32 with the solution to the first problem given in Section 3.28? You can immediately see that the solution to problem 3.32 is:

$$\mathbf{U}_K \quad \text{in} \quad (3.32) = \mathbf{U}_K \quad \text{in} \quad (3.28) \quad (3.33)$$

$$\mathbf{Z} \quad \text{in} \quad (3.32) = \Sigma_K \mathbf{V}_K^T \quad \text{in} \quad (3.28) \quad (3.34)$$

Thus, if the data points are represented by columns of a matrix and the mean of each row of that matrix is zero (so that the mean vector is zero), then the solution to PCA can be directly derived from the Truncated SVD of that matrix. In other words, the solution to PCA is a special case of Matrix Factorization solved by SVD.

### 3.6.13 How to Choose the Dimension of the New Data

One question that arises is how to choose the value of  $K$  - the dimension of the new data - for different types of data.

A way to determine  $K$  is based on the amount of information to be retained. As explained, PCA is also known as the method for maximizing total variance retained. Thus, we can consider the total retained variance as the retained information. Higher variance implies greater data dispersion, indicating more information.

Recall that in any coordinate system, the total variance of the data is equal to the sum of the eigenvalues of the covariance matrix  $\sum_{i=1}^D \lambda_i$ . Additionally, PCA retains an information amount (sum of variances) of  $\sum_{i=1}^K \lambda_i$ . Therefore, we can use the expression:

$$r_K = \frac{\sum_{i=1}^K \lambda_i}{\sum_{j=1}^D \lambda_j} \quad (3.35)$$

as the retained information when the new data dimension after PCA is  $K$ .

Thus, if we want to retain 99% of the data, we only need to choose  $K$  as the smallest integer such that  $r_K \geq 0.99$ .

When data is distributed around a subspace, the largest variance values correspond to the first few  $\lambda_i$  values, which are much larger than the remaining variances. In this case,  $K$  can be chosen quite small to achieve  $r_K \geq 0.99$ .

### 3.6.14 Notes on PCA in Real-World Applications

There are two cases to consider in real-world applications of PCA. The first is when the available data quantity is much smaller than the data dimension. The second is when the training data set is very large, possibly reaching millions, making covariance matrix and eigenvalue computation infeasible. Effective solutions exist for these cases.

In this section, we assume the data has been normalized, meaning the mean vector has been subtracted. In this case, the covariance matrix is  $\mathbf{S} = \frac{1}{N} \mathbf{XX}^T$ .

## 3.7 Data Dimension Greater than the Number of Data Points

This occurs when  $D > N$ , meaning the data matrix  $\mathbf{X}$  is a 'tall matrix'. In this case, the number of non-zero eigenvalues of the covariance matrix  $\mathbf{S}$  does not exceed its rank, i.e.,  $N$ . Thus, we must choose  $K \leq N$  because it is impossible to select  $K > N$  non-zero eigenvalues of a matrix with rank  $N$ .

Eigenvalue and eigenvector computation can be efficiently performed based on the following properties:

**Property 1:** An eigenvalue of  $\mathbf{A}$  is also an eigenvalue of  $k\mathbf{A}$  for any non-zero  $k$ . This can be directly deduced from the definitions of eigenvalues and eigenvectors.

**Property 2:** The eigenvalues of  $\mathbf{AB}$  are also eigenvalues of  $\mathbf{BA}$  where  $\mathbf{A} \in \mathbb{R}^{d_1 \times d_2}$  and  $\mathbf{B} \in \mathbb{R}^{d_2 \times d_1}$  are arbitrary matrices with non-zero dimensions  $d_1, d_2$ .

Thus, instead of finding the eigenvalues of the covariance matrix  $\mathbf{S} \in \mathbb{R}^{D \times D}$ , we find the eigenvalues of the smaller matrix  $\mathbf{T} = \mathbf{X}^T \mathbf{X} \in \mathbb{R}^{N \times N}$  (since  $N < D$ ).

**Property 3:** If  $(\lambda, \mathbf{u})$  is an eigenvalue-eigenvector pair of  $\mathbf{T}$ , then  $(\lambda, \mathbf{X}\mathbf{u})$  is an eigenvalue-eigenvector pair of  $\mathbf{S}$ .

Indeed:

$$\mathbf{X}^T \mathbf{X}\mathbf{u} = \lambda\mathbf{u} \quad (7)$$

$$\Rightarrow (\mathbf{X}\mathbf{X}^T)(\mathbf{X}\mathbf{u}) = \lambda\mathbf{X}\mathbf{u} \quad (8)$$

Expression (7) follows from the eigenvalue definition. Expression (8) is derived by multiplying both sides of (7) by  $\mathbf{X}$  on the left. This leads to **Observation 3**.

Thus, we can compute the eigenvalues and eigenvectors of the covariance matrix using a smaller matrix.

## 3.8 Large-Scale Problems

In many problems, both  $D$  and  $N$  are very large, meaning we need to find eigenvalues of a large matrix. For example, with a million images of size  $1000 \times 1000$  pixels,  $D = N = 10^6$ , which is a large number, making direct eigenvalue computation infeasible. A faster approximation method is the [Power Method](#).

The Power Method states that performing the following procedure yields the first eigenvalue-eigenvector pair of a positive semi-definite matrix:

1. Choose any vector  $\mathbf{q}^{(0)} \in \mathbb{R}^n$  with  $\|\mathbf{q}^{(0)}\|_2 = 1$ .
2. For  $k = 1, 2, \dots$ , compute:  $\mathbf{z} = \mathbf{A}\mathbf{q}^{(k-1)}$ .
3. Normalize:  $\mathbf{q}^{(k)} = \mathbf{z}/\|\mathbf{z}\|_2$ .
4. If  $\|\mathbf{q}^{(k)} - \mathbf{q}^{(k-1)}\|_2$  is sufficiently small, stop. Otherwise, increment  $k$  and return 2.
5.  $\mathbf{q}^{(k)}$  is the eigenvector corresponding to the largest eigenvalue  $\lambda_1 = (\mathbf{q}^{(k)})^T \mathbf{A} \mathbf{q}^{(k)}$ .

This process converges quickly, and the proof can be found [here](#). For the second eigenvalue-eigenvector pair, we apply the following theorem:

**Theorem:** If  $\mathbf{A}$  has eigenvalues  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$  and orthonormal eigenvectors  $\mathbf{v}_1, \dots, \mathbf{v}_n$ , then:

$$\mathbf{B} = \mathbf{A} - \lambda_1 \mathbf{v}_1 \mathbf{v}_1^T$$

has eigenvalues  $\lambda_2, \lambda_3, \dots, \lambda_n$  and corresponding eigenvectors  $\mathbf{v}_2, \mathbf{v}_3, \dots, \mathbf{v}_n, \mathbf{v}_1$ .

**The proof:**

For  $i = 1$ :

$$\mathbf{B}\mathbf{v}_1 = (\mathbf{A} - \lambda_1 \mathbf{v}_1 \mathbf{v}_1^T)\mathbf{v}_1 = \mathbf{0}$$

For  $i > 1$ :

$$\mathbf{B}\mathbf{v}_i = (\mathbf{A} - \lambda_1 \mathbf{v}_1 \mathbf{v}_1^T)\mathbf{v}_i = \mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

Thus, the theorem is proven.

This approach allows the Power Method to find (approximate) all eigenvalues and corresponding eigenvectors of the covariance matrix. It is often used in practice to find up to the  $K$ -th eigenvalue.

## 3.9 Likelihood Function and Maximum Likelihood Estimation

### 3.9.1 Likelihood Function

In logistic regression, **the likelihood function represents the probability of observing the given data under a specific set of model parameters.** It is defined as the joint probability of all observed outcomes (binary labels) given their corresponding predicted probabilities.

For logistic regression, the likelihood of a single observation  $i$  is:

$$\mathcal{L}_i = P(y_i|X_i)^{y_i} (1 - P(y_i|X_i))^{1-y_i}$$

where  $y_i \in \{0, 1\}$  is the observed class label for the  $i$ -th data point, and  $P(y_i|X_i)$  is the predicted probability of class 1.

The **total likelihood** for all observations is then the product of each individual likelihood:

$$\mathcal{L}(\beta) = \prod_{i=1}^N \mathcal{L}_i$$

### 3.9.2 Log-Likelihood Function

Directly maximizing the likelihood function (as is done in Maximum Likelihood Estimation) can be computationally challenging due to the multiplicative terms in the product. To simplify calculations, we often use the log-likelihood, which is the natural logarithm of the likelihood function. Since the logarithm is a monotonic function, maximizing the log-likelihood is equivalent to maximizing the likelihood.

The log-likelihood function for logistic regression is:

$$\ell(\beta) = \sum_{i=1}^N [y_i \log(P(y_i|X_i)) + (1 - y_i) \log(1 - P(y_i|X_i))]$$

This transformation turns the product into a sum, simplifying the process of optimization.

### 3.9.3 Maximum Likelihood Estimation

Maximum Likelihood Estimation is a method to estimate the parameters of a statistical model by maximizing the likelihood function. In logistic regression, Maximum Likelihood Estimation aims to find the values of  $\beta$  that maximize the likelihood of observing the data.

Essentially, Maximum Likelihood Estimation seeks the model parameters that make the observed data "most probable" under the logistic model.

The reason for using Maximum Likelihood Estimation is rooted in probability theory: by maximizing the likelihood, we obtain the parameter values that are most likely to have generated the observed data. This approach has strong theoretical foundations because:

- Maximum Likelihood Estimation estimators are often **consistent**, meaning they converge to the true parameter values as the sample size increases.
- They are also **efficient** and **asymptotically unbiased**, providing accurate estimates with large enough data.

### 3.9.4 Optimization in Logistic Regression

Maximizing the log-likelihood in logistic regression typically requires iterative optimization methods because the function is nonlinear and does not have a closed-form solution.

During each iteration, these algorithms adjust the model parameters (coefficients) to increase the log-likelihood until convergence. Once the log-likelihood reaches a maximum, the resulting parameter estimates are considered the Maximum Likelihood Estimates of the model coefficients.

## 3.10 Radial Basis Functions (RBFs)

### 3.10.1 Overview

A **Radial Basis Function (RBF)** is a real-valued function whose output depends only on the distance between the input point  $\mathbf{x}$  and a fixed reference point, often referred to as the **center  $\mathbf{c}$** . Formally, a function  $\varphi(\mathbf{x})$  is considered a radial basis function if it satisfies:

$$\varphi(\mathbf{x}) = \hat{\varphi}(\|\mathbf{x} - \mathbf{c}\|),$$

where  $\|\mathbf{x} - \mathbf{c}\|$  denotes the Euclidean distance between the input  $\mathbf{x}$  and the center  $\mathbf{c}$ .

#### Historical Context

Radial basis functions were first introduced in the context of multivariate interpolation by mathematician Michael J.D. Powell in the 1970s. However, their application gained prominence in machine learning in the late 1980s, particularly for function approximation and classification tasks. David Broomhead and David Lowe utilized RBFs to construct a type of neural network called the **Radial Basis Function Network (RBF Network)**, which can approximate any continuous function given enough RBFs.

#### Why the Name?

The term "Radial" refers to the fact that these functions are **radially symmetric** around their center. In other words, the **value of the function depends solely on the distance from the center and not on the direction**. This property leads to the function having the same value at any point that is equidistant from the center, forming concentric contours or "radial" shapes in space.

The phrase "Basis Function" indicates that these functions can be used as the building blocks for approximating other functions. In particular, linear combinations of RBFs are often used to construct more complex functions, much like how sine and cosine functions serve as basis functions in Fourier analysis. Thus, the combination of the terms reflects their use in forming function spaces through radially symmetric components.

### 3.10.2 Commonly Used Radial Basis Functions

Radial basis functions can be classified into various types based on their properties:

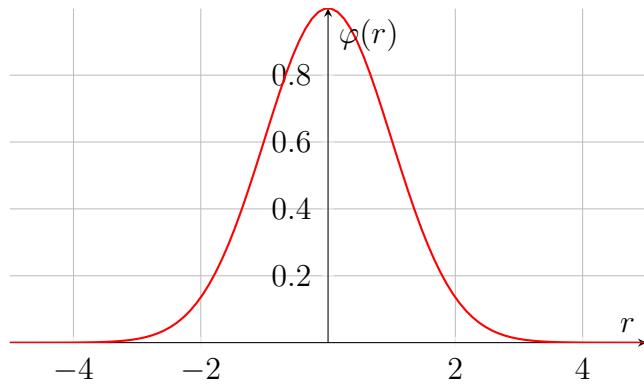
#### Infinitely Smooth Radial Basis Functions

These functions are smooth ( $C^\infty(\mathbb{R})$ ) and strictly positive definite. They typically require tuning a shape parameter  $\varepsilon$  to control the width of the basis function.

- **Gaussian RBF:**

$$\varphi(r) = e^{-(\varepsilon r)^2}$$

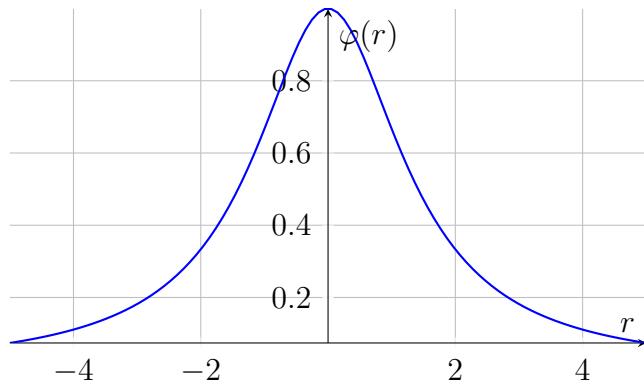
Gaussian RBF ( $\exp(\varepsilon = 0.5)$ )



- Inverse Quadratic:

$$\varphi(r) = \frac{1}{1 + (\varepsilon r)^2}$$

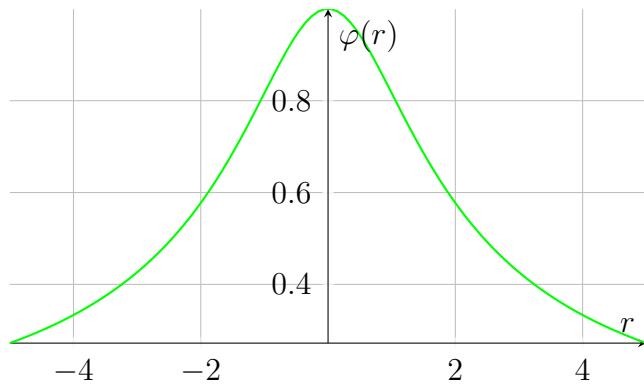
Inverse Quadratic RBF ( $\varepsilon = 0.5$ )



- Inverse Multiquadric:

$$\varphi(r) = \frac{1}{\sqrt{1 + (\varepsilon r)^2}}$$

Inverse Multiquadric RBF ( $\varepsilon = 0.5$ )

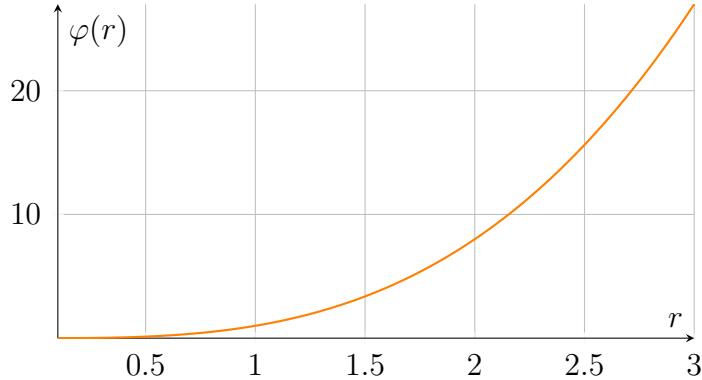


- **Polyharmonic Splines:**

$$\varphi(r) = \begin{cases} r^k, & k = 1, 3, 5, \dots \\ r^k \ln(r), & k = 2, 4, 6, \dots \end{cases}$$

Example with  $k = 3$ :

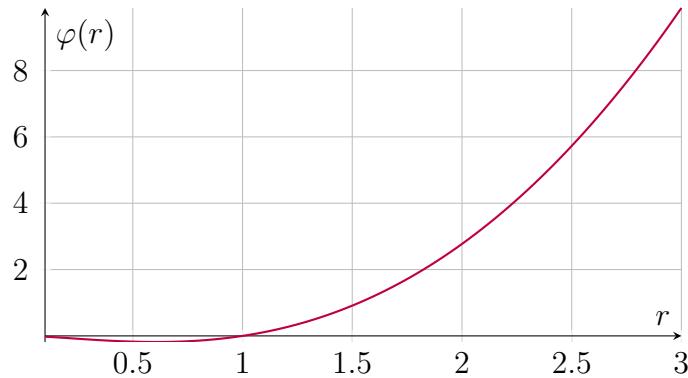
Polyharmonic Spline ( $k=3$ )



- **Thin Plate Spline (a special polyharmonic spline):**

$$\varphi(r) = r^2 \ln(r)$$

Thin Plate Spline



### Compactly Supported Radial Basis Function

Compactly supported radial basis functions have non-zero values only within a limited range. These functions are ideal for constructing sparse differentiation matrices.

#### Bump Function

The bump function is defined as:

$$\varphi(r) = \begin{cases} \exp\left(-\frac{1}{1 - (\varepsilon r)^2}\right), & \text{if } r < \frac{1}{\varepsilon} \\ 0, & \text{otherwise} \end{cases}$$

### 3.10.3 Approximation Using Radial Basis Functions

Radial basis functions are commonly used for function approximation and interpolation. The general form of an approximation using RBFs is:

$$y(\mathbf{x}) = \sum_{i=1}^N w_i \varphi(\|\mathbf{x} - \mathbf{x}_i\|),$$

where:

- $\mathbf{x}_i$  are the centers of the basis functions.
- $w_i$  are the weights to be determined.

The weights  $w_i$  can be found using linear least squares, making RBF interpolation a powerful tool for approximating continuous functions from discrete data.

### 3.10.4 Radial Basis Function Networks

An RBF network is a type of neural network that uses radial basis functions as activation functions. The output of an RBF network is given by:

$$y(\mathbf{x}) = \sum_{i=1}^N w_i \varphi(\|\mathbf{x} - \mathbf{x}_i\|),$$

where  $N$  is the number of neurons in the hidden layer. The weights  $w_i$  can be learned using standard iterative optimization techniques, such as gradient descent.

## Applications

- **Time Series Prediction:** RBF networks can be used to predict future values in time series data.
- **3D Reconstruction:** Used in computer graphics for reconstructing surfaces from scattered data points.
- **Support Vector Machines (SVMs):** RBFs are commonly used as kernels in SVMs for classification tasks.

### 3.10.5 Radial Basis Functions for Solving PDEs

Radial basis functions can be employed to discretize and numerically solve partial differential equations (PDEs). The Kansa method, developed by E.J. Kansa, was one of the first techniques to use RBFs for solving PDEs. The solution to a PDE can be approximated as:

$$u(\mathbf{x}) = \sum_{i=1}^N \lambda_i \varphi(\|\mathbf{x} - \mathbf{x}_i\|), \quad \mathbf{x} \in \mathbb{R}^d,$$

where  $\lambda_i$  are coefficients determined through boundary conditions and  $N$  is the number of collocation points.

### Derivatives of the Solution

The derivatives of the approximated solution can be calculated as:

$$\frac{\partial^n u(\mathbf{x})}{\partial x^n} = \sum_{i=1}^N \lambda_i \frac{\partial^n}{\partial x^n} \varphi(\|\mathbf{x} - \mathbf{x}_i\|).$$

### Advanced Methods

Several advanced RBF-based methods have been developed for solving PDEs:

- **RBF-FD Method:** Uses finite differences with RBFs to approximate derivatives.
- **RBF-QR Method:** Provides stable computations by reducing ill-conditioning.
- **RBF-PUM Method:** Uses partition of unity methods to enhance computational efficiency.

### 3.10.6 Conclusion

Radial Basis Functions have proven to be versatile tools in both theoretical and applied mathematics. Their ability to approximate functions, solve differential equations, and model complex systems makes them a valuable asset in various domains such as machine learning, numerical analysis, and signal processing.

## 3.11 Sigmoid Functions

**Definition:** A sigmoid function is a smooth, S-shaped function that maps any real-valued input to a range between 0 and 1. In the context of artificial neural networks, the term "sigmoid function" is commonly used as a synonym for the **logistic function**, which is defined as:

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

**Description:** The logistic sigmoid function outputs values between 0 and 1, making it useful for applications where outputs need to be interpreted as probabilities. It has an S-shaped curve (sigmoid shape), which is why it is named a sigmoid function. The logistic function has a domain of all real numbers and is bounded between 0 and 1.

**Properties:**

- **Bounded:** The output is constrained between 0 and 1.

- **Monotonic:** The function is non-decreasing.
- **Differentiable:** It is differentiable everywhere, with a smooth gradient, making it suitable for gradient-based optimization.
- **Invertible:** The inverse of the sigmoid function is known as the **logit function**, defined as:

$$\text{logit}(y) = \log\left(\frac{y}{1-y}\right)$$

- **Horizontal Asymptotes:** As  $x \rightarrow +\infty$ ,  $\sigma(x) \rightarrow 1$ , and as  $x \rightarrow -\infty$ ,  $\sigma(x) \rightarrow 0$ .
- **Symmetry:** The sigmoid function is symmetric around  $x = 0$ , satisfying  $\sigma(-x) = 1 - \sigma(x)$ .

### Derivative of the Sigmoid Function:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

This derivative is particularly useful in **backpropagation** for training neural networks, as it provides a simple way to compute gradients.

#### 3.11.1 Applications

- Commonly used as an **activation function** in artificial neural networks, especially in the output layer for binary classification tasks.
- Widely applied in fields like statistics, biology, and logistic regression due to its probabilistic interpretation.
- Used in **cumulative distribution functions (CDF)** for probability distributions, such as the error function (related to the CDF of a normal distribution) and the arctan function (related to the CDF of a Cauchy distribution).
- Found in applications involving smooth transitions, such as **audio signal processing** and **computer graphics**.

#### 3.11.2 Comparison with Other Sigmoid Functions

While the logistic sigmoid function is the most common, there are other functions that share the sigmoid shape but have different properties. Below is an overview of these functions:

- **Logistic Function:**

$$f(x) = \frac{1}{1 + e^{-x}}$$

The logistic function is widely used in neural networks, especially for binary classification tasks, due to its probabilistic output between 0 and 1.

- **Hyperbolic Tangent ( $\tanh$ ):** A shifted and scaled version of the logistic function.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The output range is between -1 and 1, making it zero-centered, which can lead to faster convergence in some cases. The hyperbolic tangent function is often preferred over the logistic function in hidden layers of neural networks as it can output negative values.

- **Arctangent Function:**

$$f(x) = \arctan(x)$$

Produces an S-shaped curve but with a slower asymptotic approach compared to the logistic function. It is zero-centered with outputs in the range  $(-\frac{\pi}{2}, \frac{\pi}{2})$ .

- **Gudermannian Function:**

$$f(x) = \text{gd}(x) = \int_0^x \frac{dt}{\cosh t} = 2 \arctan \left( \tanh \left( \frac{x}{2} \right) \right)$$

The Gudermannian function connects the circular and hyperbolic trigonometric functions without involving complex numbers.

- **Error Function ( $\text{erf}$ ):**

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Related to the cumulative distribution function of a normal distribution. It is used in statistics and probability, especially in the context of Gaussian distributions.

- **Softplus Function:** A smooth approximation to ReLU.

$$\text{softplus}(x) = \ln(1 + e^x)$$

This function is non-negative and differentiable, often used in the hidden layers of neural networks. It smooths out the sharp corners present in ReLU, reducing the risk of dead neurons.

- **Generalized Logistic Function:**

$$f(x) = (1 + e^{-x})^{-\alpha}, \quad \alpha > 0$$

This function introduces an additional parameter  $\alpha$  that controls the steepness of the curve.

- **Smoothstep Function:**

$$f(x) = \begin{cases} \left( \int_0^1 (1 - u^2)^N du \right)^{-1} \int_0^x (1 - u^2)^N du, & |x| \leq 1 \\ \text{sgn}(x), & |x| \geq 1 \end{cases}$$

where  $N \in \mathbb{Z} \geq 1$ . The smoothstep function is used for smooth transitions between two values and is common in computer graphics.

- **Algebraic Function:**

$$f(x) = \frac{x}{\sqrt{1+x^2}}$$

This function is a sigmoid-like function that maps inputs to the range (-1, 1).

- **General Form of Algebraic Function:**

$$f(x) = \frac{x}{(1+|x|^k)^{1/k}}$$

This generalized form includes a parameter  $k$  that adjusts the steepness of the transition.

- **Smooth Transition Function (normalized to (-1, 1)):**

$$f(x) = \begin{cases} \frac{2}{1+e^{-2m\frac{x}{1-x^2}}}-1, & |x| < 1 \\ \text{sgn}(x), & |x| \geq 1 \end{cases}$$

or equivalently,

$$f(x) = \begin{cases} \tanh\left(m\frac{x}{1-x^2}\right), & |x| < 1 \\ \text{sgn}(x), & |x| \geq 1 \end{cases}$$

Here,  $m$  is a parameter that controls the slope at  $x = 0$ . This function is smooth and differentiable everywhere, including at  $x = \pm 1$ .

**Notes:** - These functions are all sigmoid-like, meaning they exhibit an S-shaped curve. - Some of these functions, like the hyperbolic tangent and logistic function, are widely used as activation functions in neural networks. - Other functions, such as the error function and Gudermannian function, are more common in mathematical and statistical contexts.

### 3.11.3 Pros of Using the Sigmoid Function

- Outputs values between 0 and 1, making it ideal for probabilistic interpretation.
- Smooth and differentiable, enabling gradient-based optimization.
- Useful in scenarios where binary classification or a smooth transition is needed.

### 3.11.4 Cons of Using the Sigmoid Function

- **Vanishing Gradient Problem:** For large positive or negative inputs, the gradient approaches zero, which can slow down the learning process in deep networks.

- **Not Zero-Centered:** The output range (0, 1) is not zero-centered, which can lead to slow convergence during training due to non-zero mean activations.
- **Computationally Expensive:** Involves the exponential function, which is more computationally intensive than simpler activation functions like ReLU.

## 3.12 Gradient Descent

### 3.12.1 Introduction

You might find the following image familiar:

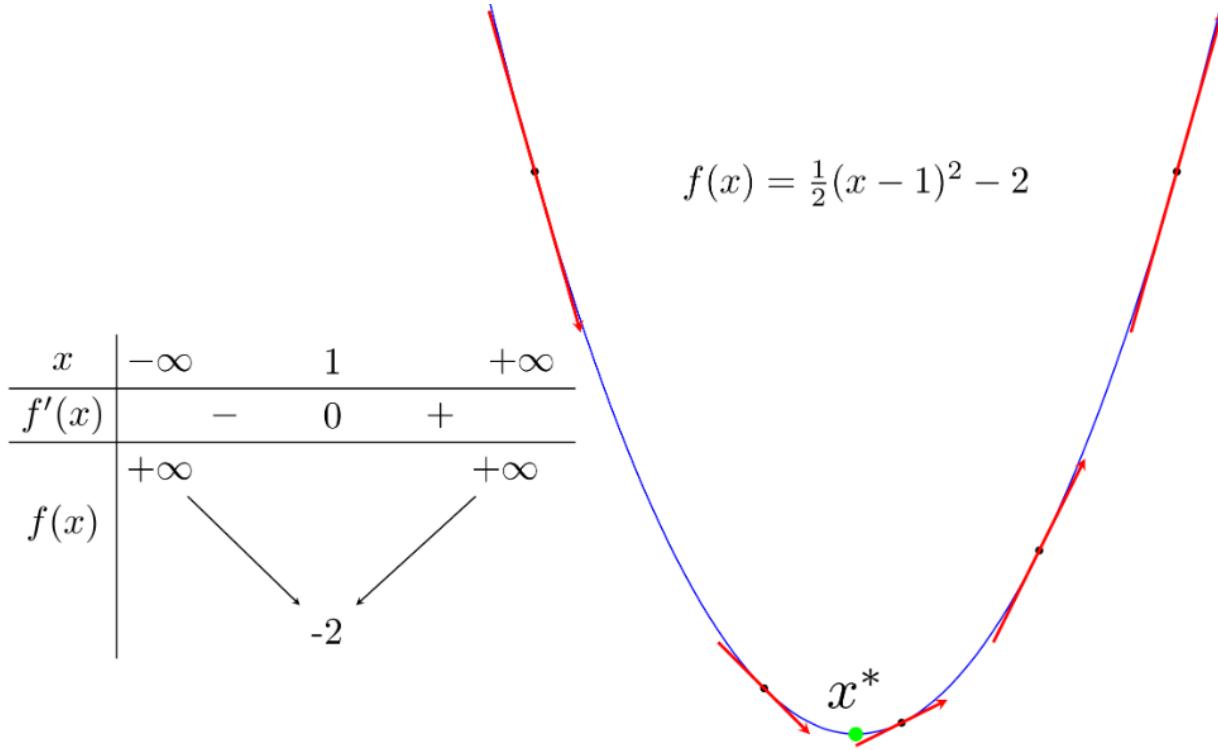


Figure 3.18: Gradient Descent: Introduction

The green point is a local minimum, which is also the point where the function reaches its smallest value. From now on, I will use *local minimum* to refer to *the point of minimum*, and *global minimum* to refer to *the point where the function reaches its smallest value*. A global minimum is a special case of a local minimum.

Assume we are interested in a single-variable function that is differentiable everywhere. Let me remind you of a few familiar facts:

1. A local minimum  $x^*$  of a function is a point where the derivative  $f'(x^*)$  is zero. Furthermore, in its vicinity, the derivative of points to the left of  $x^*$  is non-positive, and the derivative of points to the right of  $x^*$  is non-negative.
2. The tangent line to the graph of the function at any point has a slope equal to the derivative of the function at that point.

In the image above, points to the left of the green local minimum have negative derivatives, and points to the right have positive derivatives. For this function, the further left from the

local minimum, the more negative the derivative, and the further right, the more positive the derivative.

In Machine Learning and Optimization, we often need to find the minimum (or sometimes the maximum) of a function. Generally, finding the global minimum of loss functions in Machine Learning is very complex, even impossible. Instead, people often try to find local minima and, to some extent, consider them the solution to the problem.

Local minima are solutions to the equation where the derivative is zero. If we can somehow find all (finite) local minima, we only need to substitute each local minimum into the function and find the point that makes the function smallest (*this sounds familiar, doesn't it?*). However, in most cases, solving the equation where the derivative is zero is impossible. This can be due to the complexity of the derivative's form, the high dimensionality of the data points, or the sheer number of data points.

The most common approach is to start from a point we consider *close* to the solution and then use an iterative process to *gradually* reach the desired point, i.e., until the derivative is close to zero. Gradient Descent (abbreviated as GD) and its variants are among the most widely used methods.

Since the knowledge about GD is quite extensive, I will divide it into two parts. Part 1 introduces the idea behind the GD algorithm and a few simple examples to help you get familiar with this algorithm and some new concepts. Part 2 will discuss improved GD methods and GD variants in problems with high dimensionality and large data points, known as *large-scale* problems.

### 3.12.2 Gradient Descent for Single Variable Functions

Returning to the initial image and some observations I mentioned. Suppose  $x_t$  is the point found after the  $t$ -th iteration. We need an algorithm to bring  $x_t$  as close to  $x^*$  as possible.

In the first image, we have two more observations:

1. If the derivative of the function at  $x_t$ :  $f'(x_t) > 0$ , then  $x_t$  is to the right of  $x^*$  (and vice versa). To make the next point  $x_{t+1}$  closer to  $x^*$ , we need to move  $x_t$  to the left, i.e., in the *negative* direction. In other words, **we need to move in the opposite direction of the derivative**:  $x_{t+1} = x_t + \Delta$  Where  $\Delta$  is a quantity opposite in sign to the derivative  $f'(x_t)$ .
2. The further  $x_t$  is from  $x^*$  to the right, the larger  $f'(x_t)$  is (and vice versa). Thus, the movement  $\Delta$ , in the most intuitive way, is proportional to  $-f'(x_t)$ .

The two observations above give us a simple update rule:  $x_{t+1} = x_t - \eta f'(x_t)$

Where  $\eta$  (read as *eta*) is a positive number called the *learning rate*. The minus sign indicates that we must *go against* the derivative (This is also why this method is called Gradient Descent - *descent* means *going against*). These simple observations, although not true for

all problems, are the foundation for many optimization methods and Machine Learning algorithms.

### 3.12.3 Simple Example with Python

Consider the function  $f(x) = x^2 + 5 \sin(x)$  with the derivative  $f'(x) = 2x + 5 \cos(x)$  (one reason I chose this function is that it is not easy to find the solution of the derivative equal to zero like the function above). Suppose we start from a point  $x_0$ , at the  $t$ -th iteration, we update as follows:  $x_{t+1} = x_t - \eta(2x_t + 5 \cos(x_t))$

As usual, I declare a few familiar libraries:

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import math
import numpy as np
import matplotlib.pyplot as plt
```

Next, I write the functions:

1. `grad` to calculate the derivative
2. `cost` to calculate the function's value. This function is not used in the algorithm but is often used to check if the derivative is calculated correctly or to see if the function's value decreases with each iteration.
3. `myGD1` is the main part that implements the Gradient Descent algorithm mentioned above. The input to this function is the learning rate and the starting point. The algorithm stops when the derivative's magnitude is small enough.

```
1 def grad(x):
2     return 2*x+ 5*np.cos(x)
3
4 def cost(x):
5     return x**2 + 5*np.sin(x)
6
7 def myGD1(eta, x0):
8     x = [x0]
9     for it in range(100):
10        x_new = x[-1] - eta*grad(x[-1])
11        if abs(grad(x_new)) < 1e-3:
12            break
13        x.append(x_new)
14    return (x, it)
```

### 3.12.4 Different Starting Points

After having the necessary functions, I try to find solutions with different starting points  $x_0 = -5$  and  $x_0 = 5$ .

```

(x1, it1) = myGD1(.1, -5)
(x2, it2) = myGD1(.1, 5)
print('Solution x1 = %f, cost = %f,
      obtained after %d iterations',%(x1[-1], cost(x1[-1]), it1))
print('Solution x2 = %f, cost = %f,
      obtained after %d iterations',%(x2[-1], cost(x2[-1]), it2))

Solution    x1 = -1.110667,
            cost = -3.246394,
            obtained after 11 iterations
Solution    x2 = -1.110341,
            cost = -3.246394,
            obtained after 29 iterations

```

Thus, with different initial points, our algorithm finds similar solutions, although with different convergence speeds. Below is an illustration of the GD algorithm for this problem (*best viewed on Desktop in full-screen mode*).

From the illustration above, we see that in the left image, corresponding to  $x_0 = -5$ , the solution converges faster because the initial point  $x_0$  is closer to the solution  $x^* \approx -1$ . Moreover, with  $x_0 = 5$  in the right image, the *path* of the solution contains an area with a relatively small derivative near the point with an abscissa of 2. This causes the algorithm to *linger* there for quite a while. Once it passes this point, everything goes smoothly.

### 3.12.5 Different Learning Rates

The convergence speed of GD depends not only on the initial point but also on the *learning rate*. Below is an example with the same initial point  $x_0 = -5$  but different learning rates:

We observe two things:

1. With a small *learning rate*  $\eta = 0.01$ , the convergence speed is very slow. In this example, I chose a maximum of 100 iterations, so the algorithm stops before reaching the *destination*, although it is very close. In practice, when calculations become complex, a too-low *learning rate* will significantly affect the algorithm's speed, even preventing it from ever reaching the destination.
2. With a large *learning rate*  $\eta = 0.5$ , the algorithm quickly approaches the *destination* after a few iterations. However, the algorithm does not converge because the *step size* is too large, causing it to *circle around* the destination.

**Note:** Choosing the *learning rate* is crucial in real-world problems. The choice of this value depends heavily on each problem and requires some experimentation to select the best value. Additionally, depending on some problems, GD can work more effectively by choosing an appropriate *learning rate* or selecting different *learning rates* at each iteration. I will return to this issue in part 2.

### 3.12.6 Algorithm Steps

The steps in the Gradient Descent algorithm are as follows:

1. **Initialize Parameters:** Start with an initial guess for the parameters, denoted as  $\theta$ , that minimize the objective function  $f(\theta)$ . This could be any starting point, often chosen randomly.
2. **Compute the Gradient:** At the current parameter values, calculate the gradient (partial derivatives) of  $f(\theta)$  with respect to  $\theta$ . The gradient vector points in the direction of the steepest increase.
3. **Update Parameters:** Update the parameters by moving in the opposite direction of the gradient by a factor of the learning rate  $\alpha$ :

$$\theta = \theta - \alpha \nabla f(\theta)$$

$\nabla f(\theta)$  is the gradient of the function at  $\theta$  and  $\alpha$  is a small positive number.

4. **Iterate Until Convergence:** Repeat steps 2 and 3 until the change in the function value  $f(\theta)$  is smaller than a chosen threshold, or a maximum number of iterations is reached.

### Key Parameters

- **Learning Rate ( $\alpha$ ):** This controls the step size during the update. A small learning rate makes convergence slow but more stable, while a large learning rate might cause overshooting or divergence.
- **Convergence Criteria:** Gradient Descent iterates until the function's value change is negligible (below a threshold) or until a fixed number of steps is completed.

### 3.12.7 Types of Gradient Descent

Gradient Descent has three main variants, depending on how often gradients are calculated:

1. **Batch Gradient Descent:** Calculates the gradient using the entire dataset. It is computationally expensive but offers a stable descent path.
2. **Stochastic Gradient Descent:** Calculates the gradient using one data point at a time. This is less stable but faster and can help in escaping local minima.
3. **Mini-Batch Gradient Descent:** A hybrid of Batch and Stochastic Gradient Descent, it calculates the gradient on small subsets (mini-batches) of data, balancing stability and computational efficiency.

### 3.12.8 Applications Beyond Machine Learning

Gradient Descent is widely used in:

- **Physics:** For optimizing physical models and simulations.
- **Economics:** To find optimal parameters in cost functions and utility functions.
- **Engineering:** For minimizing error in control systems or optimizing structural parameters.

### 3.12.9 Advantages and Disadvantages

**Advantages:**

- **Simple and Versatile:** Gradient Descent is straightforward to implement and applicable to a wide range of optimization problems.
- **Efficient for Large Problems:** Especially in mini-batch or stochastic variants, it can handle large datasets and complex functions.

**Disadvantages:**

- **Sensitive to Learning Rate:** Choosing an appropriate learning rate can be challenging.
- **May Converge to Local Minima:** Particularly in non-convex functions, it can get stuck in local minima (though variations like Stochastic Gradient Descent help mitigate this).

## 3.13 Stochastic Gradient Descent

Stochastic Gradient Descent is a variation of the Gradient Descent algorithm that calculates the gradient based on a single randomly chosen data point (or sample) rather than using the entire dataset. This approach speeds up computation, especially for large datasets, but introduces noise into the parameter updates, causing them to be more random and potentially unstable. However, this randomness can also help Stochastic Gradient Descent avoid local optima, making it popular in training complex machine learning models, especially deep neural networks.

### 3.13.1 How Stochastic Gradient Descent Works

In Stochastic Gradient Descent, instead of calculating the gradient of the objective function  $f(\theta)$  using all training samples, it calculates the gradient for a single data point  $x^{(i)}$  and updates the parameters accordingly. The update rule is given by:

$$\theta = \theta - \alpha \nabla f(\theta; x^{(i)}) \quad (3.38)$$

where:

- $\theta$  represents the parameters to be optimized,
- $\alpha$  is the learning rate,
- $x^{(i)}$  is a randomly selected data sample, and
- $\nabla f(\theta; x^{(i)})$  is the gradient of  $f$  with respect to  $\theta$  for sample  $x^{(i)}$ .

By updating based on a single sample, Stochastic Gradient Descent is much faster and can start to learn from the data without waiting to process the entire dataset. However, the inherent noise from using a single sample's gradient can cause Stochastic Gradient Descent to oscillate around the minimum rather than converge smoothly.

### 3.13.2 Challenges of Stochastic Gradient Descent

Stochastic Gradient Descent faces several challenges, including:

- **Convergence Stability:** Due to the noisy updates, the path to convergence is less stable and may oscillate, especially with high learning rates.
- **Local Optima:** While stochasticity helps in escaping some local minima, complex non-convex functions may contain multiple problematic local optima or saddle points that can trap the optimization process.

- **Learning Rate Sensitivity:** The choice of learning rate is crucial. A high learning rate can cause divergence or oscillation, while a low learning rate may lead to slow convergence.

### 3.13.3 Methods to Improve Stochastic Gradient Descent Performance and Avoid Local Minimum

To optimize Stochastic Gradient Descent and avoid local mininum or saddle points, various techniques have been developed:

**Learning Rate Scheduling** Adjusting the learning rate during training can help improve convergence.

- **Learning Rate Decay:** Gradually decreasing the learning rate as training progresses allows Stochastic Gradient Descent to take large steps initially and smaller, more precise steps as it approaches an optimum. Common decay strategies include:
  - *Step Decay:* Reduce the learning rate by a fixed factor at regular intervals.
  - *Exponential Decay:* Multiply the learning rate by a constant factor after each epoch.
  - *Inverse Scaling:* Reduce the learning rate proportional to the inverse of the epoch number, defined as:

$$\alpha_t = \frac{\alpha_0}{1 + \lambda t} \quad (3.39)$$

- **Adaptive Learning Rates:** Methods such as AdaGrad, RMSProp, and Adam adjust the learning rate adaptively for each parameter. These methods use information from previous gradients to scale the learning rate, speeding up convergence and helping Stochastic Gradient Descent navigate complex landscapes.

**Momentum** Momentum is a technique that helps Stochastic Gradient Descent build up speed in directions with consistent gradients, smoothing out noisy updates. The momentum update rule is given by:

$$v_t = \beta v_{t-1} + \alpha \nabla f(\theta; x^{(i)}) \quad (3.40)$$

$$\theta = \theta - v_t \quad (3.41)$$

where:

- $v_t$  is the velocity term (accumulated gradient),
- $\beta$  is the momentum coefficient (typically between 0.5 and 0.9).

By using momentum, Stochastic Gradient Descent retains part of the previous update direction, effectively accelerating convergence and reducing oscillations around the minima.

**Batch Normalization** Batch normalization is a technique that normalizes the input of each layer in neural networks to have zero mean and unit variance. This normalization can help the optimization landscape become smoother and more consistent across different regions, reducing the likelihood of getting trapped in local optima or saddle points.

**Gradient Noise Injection** Adding random noise to gradients can prevent the optimizer from getting trapped in sharp local minima. This noise acts as a perturbation, potentially allowing the algorithm to escape from narrow local minima. The update rule with noise is:

$$\theta = \theta - \alpha \nabla f(\theta; x^{(i)}) + \epsilon \quad (3.42)$$

where  $\epsilon$  is random noise sampled from a Gaussian or uniform distribution. The noise level can be reduced over time to help converge near a minimum.

### Restarts and Ensembles

- **Warm Restarts:** A technique where the learning rate is periodically reset to a high value to help escape local optima, then reduced again. This approach is used in techniques such as Cosine Annealing and Stochastic Gradient Descent with Restarts.
- **Ensemble Methods:** Running multiple Stochastic Gradient Descent instances with different initializations or slightly different parameters (like learning rate) increases the chances of finding a global or near-global minimum. Aggregating solutions from each instance can lead to better overall performance.

**Variants and Alternatives to Stochastic Gradient Descent** Several Stochastic Gradient Descent-based optimizers address specific challenges and can perform better in complex landscapes:

- **Adam (Adaptive Moment Estimation):** Combines ideas from momentum and adaptive learning rates, adjusting the step size based on an exponentially decaying average of past gradients and squared gradients. Adam has shown robust performance and is less sensitive to hyperparameter tuning.
- **Nesterov Accelerated Gradient (NAG):** An improvement over basic momentum, Nesterov momentum calculates the gradient after applying the velocity, which gives a “look-ahead” and prevents overshooting.

## 3.14 Gradient Descent for Multi-variable Functions

Suppose we need to find the global minimum for the function  $f(\theta)$  where  $\theta$  (*theta*) is a vector, often used to denote the set of parameters of a model to be optimized (in Linear Regression, the parameters are the coefficients  $\mathbf{w}$ ). The derivative of the function at any point  $\theta$  is denoted as  $\nabla_{\theta}f(\theta)$  (the inverted triangle is read as *nabla*). Similar to single-variable functions, the GD algorithm for multi-variable functions also starts with an initial guess  $\theta_0$ , and at the  $t$ -th iteration, the update rule is:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} f(\theta_t)$$

Or written more simply:  $\theta = \theta - \eta \nabla_{\theta} f(\theta)$ .

The rule to remember: **always go in the opposite direction of the derivative.**

Calculating derivatives of multi-variable functions is a necessary skill. Some simple derivatives can be [found here](#).

### 3.14.1 Back to Linear Regression

In this section, we return to the [Linear Regression](#) problem and try to optimize its loss function using the GD algorithm.

The loss function of Linear Regression is:  $\mathcal{L}(\mathbf{w}) = \frac{1}{2N} \|\|\mathbf{y} - \bar{\mathbf{X}}\mathbf{w}\|\|^2_2$

**Note:** this function is slightly different from the one I mentioned in the [Linear Regression](#) article. The denominator includes  $N$ , the number of data points in the training set. Averaging the error helps avoid cases where the loss function and derivative have very large values, affecting the accuracy of calculations on computers. Mathematically, the solutions to the two problems are the same.

The derivative of the loss function is:  $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{N} \bar{\mathbf{X}}^T (\bar{\mathbf{X}}\mathbf{w} - \mathbf{y}) \quad (1)$

### 3.14.2 Example in Python and Some Programming Notes

Load libraries

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
np.random.seed(2)
```

Next, we create 1000 data points chosen *close* to the line  $y = 4 + 3x$ , display them, and find the solution using the formula:

```
X = np.random.rand(1000, 1)
```

```

2 y = 4 + 3 * X + .2*np.random.randn(1000, 1) # noise added
3
4 # Building Xbar
5 one = np.ones((X.shape[0],1))
6 Xbar = np.concatenate((one, X), axis = 1)
7
8 A = np.dot(Xbar.T, Xbar)
9 b = np.dot(Xbar.T, y)
10 w_lr = np.dot(np.linalg.pinv(A), b)
11 print('Solution found by formula: w = ',w_lr.T)
12
13 # Display result
14 w = w_lr
15 w_0 = w[0][0]
16 w_1 = w[1][0]
17 x0 = np.linspace(0, 1, 2, endpoint=True)
18 y0 = w_0 + w_1*x0
19
20 # Draw the fitting line
21 plt.plot(X.T, y.T, 'b.')      # data
22 plt.plot(x0, y0, 'y', linewidth = 2)    # the fitting line
23 plt.axis([0, 1, 0, 10])
24 plt.show()

```

Solution found by formula: w = [[ 4.00305242 2.99862665]]

Next, we write the derivative and loss function:

```

def grad(w):
    N = Xbar.shape[0]
    return 1/N * Xbar.T.dot(Xbar.dot(w) - y)

def cost(w):
    N = Xbar.shape[0]
    return .5/N*np.linalg.norm(y - Xbar.dot(w), 2)**2;

```

### 3.14.3 Checking the Derivative

Calculating derivatives of multi-variable functions is usually quite complex and prone to errors. If we calculate the derivative incorrectly, the GD algorithm cannot run correctly. In practice, there is a way to check whether the calculated derivative is accurate. This method is based on the definition of the derivative (for single-variable functions):  $f'(x) = \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon}$

A commonly used method is to take a very small value  $\varepsilon$ , for example,  $10^{-6}$ , and use the formula:  $f'(x) \approx \frac{f(x + \varepsilon) - f(x - \varepsilon)}{2\varepsilon}$  (2)

This method is called the *numerical gradient*.

**Question:** Why is the two-sided approximation formula above widely used, and why not use the right or left derivative approximation?

There are two explanations for this issue, one geometric and one analytic.

### 3.14.4 Geometric Explanation

Observe the image below:

In the image, the red vector is the *exact* derivative of the function at the point with abscissa  $x_0$ . The blue vector (which appears slightly purple after converting from .pdf to .png) represents the right derivative approximation. The green vector represents the left derivative approximation. The brown vector represents the two-sided derivative approximation. Among these approximations, the two-sided brown vector is closest to the red vector in terms of direction.

The difference between the approximations becomes even more significant if the function is *bent* more strongly at point  $x$ . In that case, the left and right approximations will differ significantly. The two-sided approximation will be more *stable*.

### 3.14.5 Analytic Explanation

Let's revisit a bit of first-year university Calculus I: Taylor Series Expansion.

With a very small  $\varepsilon$ , we have two approximations:

$$f(x + \varepsilon) \approx f(x) + f'(x)\varepsilon + \frac{f''(x)}{2}\varepsilon^2 + \dots$$

$$\text{and: } f(x - \varepsilon) \approx f(x) - f'(x)\varepsilon + \frac{f''(x)}{2}\varepsilon^2 - \dots$$

$$\text{From this, we have: } \frac{f(x + \varepsilon) - f(x)}{\varepsilon} \approx f'(x) + \frac{f''(x)}{2}\varepsilon + \dots = f'(x) + O(\varepsilon) \quad (3)$$

$$\frac{f(x + \varepsilon) - f(x - \varepsilon)}{2\varepsilon} \approx f'(x) + \frac{f^{(3)}(x)}{6}\varepsilon^2 + \dots = f'(x) + O(\varepsilon^2) \quad (4)$$

From this, if the derivative is approximated using formula (3) (right derivative approximation), the error will be  $O(\varepsilon)$ . Meanwhile, if the derivative is approximated using formula (4) (two-sided derivative approximation), the error will be  $O(\varepsilon^2) \ll O(\varepsilon)$  if  $\varepsilon$  is small.

Both explanations above show that the two-sided derivative approximation is a better approximation.

### 3.14.6 For Multi-variable Functions

For multi-variable functions, formula (2) is applied to each variable while keeping the other variables fixed. This method usually provides quite accurate values. However, this method is not used to calculate derivatives due to its high complexity compared to direct calculation. When comparing this derivative with the exact derivative calculated using the formula, people often reduce the dimensionality of the data and the number of data points to facilitate

calculations. Once the calculated derivative is very close to the *numerical gradient*, we can be confident that the calculated derivative is accurate.

Below is a simple code snippet to check the derivative, which can be applied to any function (of a vector) with the `cost` and `grad` calculated above.

```

1 def numerical_grad(w, cost):
2     eps = 1e-4
3     g = np.zeros_like(w)
4     for i in range(len(w)):
5         w_p = w.copy()
6         w_n = w.copy()
7         w_p[i] += eps
8         w_n[i] -= eps
9         g[i] = (cost(w_p) - cost(w_n))/(2*eps)
10    return g
11
12 def check_grad(w, cost, grad):
13     w = np.random.rand(w.shape[0], w.shape[1])
14     grad1 = grad(w)
15     grad2 = numerical_grad(w, cost)
16     return True if np.linalg.norm(grad1 - grad2) < 1e-6 else False
17
18 print('Checking gradient...', check_grad(np.random.rand(2, 1), cost, grad))

```

```
Checking gradient... True
```

For other functions, readers only need to rewrite the `grad` and `cost` functions above and apply this code snippet to check the derivative. If the function is a function of a matrix, we need to make a slight change in the `numerical_grad` function, which I hope is not too complicated.

For the Linear Regression problem, the derivative calculation as in (1) above is considered correct because the error between the two calculations is very small (less than  $10^{-6}$ ). After obtaining the correct derivative, we write the GD function:

```

def myGD(w_init, grad, eta):
    w = [w_init]
    for it in range(100):
        w_new = w[-1] - eta*grad(w[-1])
        if np.linalg.norm(grad(w_new))/len(w_new) < 1e-3:
            break
        w.append(w_new)
    return (w, it)

w_init = np.array([[2], [1]])
(w1, it1) = myGD(w_init, grad, 1)
print('Solution found by GD: w = ', w1[-1].T, ',\nafter %d iterations.' %(it1+1))

```

```
Solution found by GD: w =  [[ 4.01780793   2.97133693]] ,
after 49 iterations.
```

After 49 iterations, the algorithm converges with a solution quite close to the solution found using the formula.

Below is an animation illustrating the GD algorithm.

In the left image, the red lines are the solutions found after each iteration.

In the right image, I introduce a new term: *level sets*.

### 3.14.7 Level Sets

For the graph of a function with two input variables to be drawn in three-dimensional space, it is often difficult to see the approximate coordinates of the solution. In optimization, people often use a drawing method that uses the concept of *level sets*.

If you pay attention to natural maps, to describe the height of mountain ranges, people use many closed curves surrounding each other as follows:

The smaller red circles represent points at higher altitudes.

In optimization, this method is also used to represent surfaces in two-dimensional space.

Returning to the GD algorithm illustration for the Linear Regression problem above, the right image represents the level sets. That is, at points on the same circle, the loss function has the same value. In this example, I display the function's value at several circles. The green circles have low values, and the outer red circles have higher values. This is slightly different from natural level sets, where the inner circles usually represent a valley rather than a mountain peak (because we are looking for the smallest value).

I try with a smaller *learning rate*, and the result is as follows:

The convergence speed has slowed significantly, and even after 99 iterations, GD has not yet reached the best solution. In real-world problems, we need many more iterations than 99 because the number of dimensions and data points is usually very large.

### Another Example

To conclude part 1 of Gradient Descent, I present another example.

The function  $f(x, y) = (x^2 + y - 7)^2 + (x - y + 1)^2$  has two green local minima at  $(2, 3)$  and  $(-3, -2)$ , which are also two global minima. In this example, depending on the initial point, we obtain different final solutions.

## 3.15 Optimization Problems

In Optimization, a constrained optimization problem is often written in the following form:

$$\begin{aligned}\mathbf{x}^* &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } &f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ &h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots, p\end{aligned}$$

Here, the vector  $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$  is called the *optimization variable*. The function  $f_0 : \mathbb{R}^n \rightarrow \mathbb{R}$  is referred to as the *objective function* (in Machine Learning, objective functions are often referred to as loss functions). The functions  $f_i, h_j : \mathbb{R}^n \rightarrow \mathbb{R}, i = 1, 2, \dots, m; j = 1, 2, \dots, p$  are the *constraint functions* (or simply constraints). The set of points  $\mathbf{x}$  that satisfy the constraints is called the *feasible set*. Any point in the feasible set is referred to as a *feasible point*, while those not in the feasible set are called *infeasible points*.

**Notes:**

- If the problem is to find the maximum instead of the minimum, we simply negate  $f_0(\mathbf{x})$ .
- If the constraint is " $\geq$ ", i.e.,  $f_i(\mathbf{x}) \geq b_i$ , we can negate the constraint to get " $\leq$ " by writing  $-f_i(\mathbf{x}) \leq -b_i$ .
- Constraints can also be strict inequalities.
- An equality constraint, i.e.,  $h_j(\mathbf{x}) = 0$ , can be written as two inequalities  $h_j(\mathbf{x}) \leq 0$  and  $-h_j(\mathbf{x}) \leq 0$ . In some texts, equality constraints are omitted.
- In this text,  $\mathbf{x}, \mathbf{y}$  are mainly used to denote variables, not data as in previous discussions. The optimization variable is the one shown in the  $\arg \min$  notation.

In general, optimization problems do not have a universal solution method, and some problems are still unsolved. Most solution methods cannot guarantee that the solution is the global optimum, i.e., the true minimum or maximum. Instead, the solution is often a local optimum, i.e., a local extremum.

## 3.16 Duality

We firstly take a look at a single-constraint problem:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } &f_1(\mathbf{x}) = 0 \end{aligned} \quad (3.43)$$

### 3.16.1 Lagrange Multiplier Method

If we can turn this problem into an unconstrained optimization problem, we can find the solution by solving a system of equations where the partial derivatives are equal to zero (assuming that solving such a system is feasible).

This was the motivation for the mathematician Lagrange to use the function:

$$\mathcal{L}(\mathbf{x}, \lambda) = f_0(\mathbf{x}) + \lambda f_1(\mathbf{x})$$

Note that, in this function, we have an additional variable  $\lambda$ , called the Lagrange multiplier. The function  $\mathcal{L}(\mathbf{x}, \lambda)$  is known as the *auxiliary function*, or the *Lagrangian*. It has been proven that the optimal value of problem (1) satisfies the condition  $\nabla_{\mathbf{x}, \lambda} \mathcal{L}(\mathbf{x}, \lambda) = 0$  (the proof of this is omitted here). This is equivalent to:

$$\nabla_{\mathbf{x}} f_0(\mathbf{x}) + \lambda \nabla_{\mathbf{x}} f_1(\mathbf{x}) = 0 \quad (3.44)$$

$$f_1(\mathbf{x}) = 0 \quad (3.45)$$

Note that the second condition is equivalent to  $\nabla_{\lambda} \mathcal{L}(\mathbf{x}, \lambda) = 0$ , which is also the constraint in problem (3.44).

Solving the system of equations (3.44)–(3.45) is, in many cases, simpler than directly finding the *optimal value* of problem (3.44).

Consider the following simple examples.

**Example 1:** Find the maximum and minimum values of the function  $f_0(x, y) = x + y$  subject to the condition  $f_1(x, y) = x^2 + y^2 = 2$ . Note that this is not a convex optimization problem because the *feasible set*  $x^2 + y^2 = 2$  is not a convex set (it is just a circle).

Solution:

The *Lagrangian* of this problem is:

$$\mathcal{L}(x, y, \lambda) = x + y + \lambda(x^2 + y^2 - 2)$$

The extremum points of the Lagrange function must satisfy the condition:

$$\nabla_{x,y,\lambda} \mathcal{L}(x, y, \lambda) = 0 \Leftrightarrow \begin{cases} 1 + 2\lambda x = 0 \\ 1 + 2\lambda y = 0 \\ x^2 + y^2 = 2 \end{cases}$$

We get  $x = y = \frac{-1}{2\lambda}$  from the above set of equations. Substituting into  $x^2 + y^2 = 2$ , we have  $\lambda^2 = \frac{1}{4} \Rightarrow \lambda = \pm\frac{1}{2}$ . Thus, we have two pairs of solutions  $(x, y) \in \{(1, 1), (-1, -1)\}$ . By substituting these values into the objective function, we can find the minimum and maximum values of the function.

**Example 2: Cross-entropy.** We introduced the loss function in the form of cross-entropy. The cross-entropy function is used to measure the similarity between two probability distributions, where the smaller the value of the function, the closer the distributions are. The minimum value of the cross-entropy is achieved when the two probability distributions are identical. Let's prove this assertion.

Consider a probability distribution  $\mathbf{p} = [p_1, p_2, \dots, p_n]^T$  with  $p_i \in [0, 1]$  and  $\sum_{i=1}^n p_i = 1$ . For an arbitrary probability distribution  $\mathbf{q} = [q_1, q_2, \dots, q_n]$ , assuming  $q_i \neq 0, \forall i$ , the cross-entropy function is defined as:

$$f_0(\mathbf{q}) = -\sum_{i=1}^n p_i \log(q_i)$$

We need to find  $\mathbf{q}$  such that the cross-entropy is minimized.

In this problem, the constraint is  $\sum_{i=1}^n q_i = 1$ . The *Lagrangian* of the problem is:

$$\mathcal{L}(q_1, q_2, \dots, q_n, \lambda) = -\sum_{i=1}^n p_i \log(q_i) + \lambda \left( \sum_{i=1}^n q_i - 1 \right)$$

We need to solve the following system of equations:

$$\nabla_{q_1, \dots, q_n, \lambda} \mathcal{L}(q_1, \dots, q_n, \lambda) = 0 \Leftrightarrow \begin{cases} -\frac{p_i}{q_i} + \lambda &= 0, \quad i = 1, \dots, n \\ q_1 + q_2 + \dots + q_n &= 1 \end{cases}$$

From the above equation, we can conclude  $p_i = \lambda q_i$ .

Therefore:  $1 = \sum_{i=1}^n p_i = \lambda \sum_{i=1}^n q_i = \lambda \Rightarrow \lambda = 1 \Rightarrow q_i = p_i, \forall i$ .

This shows why the cross-entropy function is used to *force* two probability distributions to be *close*.

### 3.16.2 The Lagrange Dual Function

#### Lagrangian

For a general optimization problem:

$$\begin{aligned} \mathbf{x}^* &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } &f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ &h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots, p \end{aligned} \tag{3.46}$$

with the domain  $\mathcal{D} = (\cap_{i=0}^m \text{dom } f_i) \cap (\cap_{j=1}^p \text{dom } h_j)$ . Note that we are not assuming convexity of the objective function or the constraints here. The only assumption is that  $\mathcal{D} \neq \emptyset$  (non-empty set).

The *Lagrangian* is constructed similarly with a Lagrange multiplier for each (in)equality constraint:

$$\mathcal{L}(\mathbf{x}, \lambda, \nu) = f_0(\mathbf{x}) + \sum_{i=1}^m \lambda_i f_i(\mathbf{x}) + \sum_{j=1}^p \nu_j h_j(\mathbf{x})$$

where  $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_m]; \nu = [\nu_1, \nu_2, \dots, \nu_p]$  (note that  $\nu$  is the Greek letter *nu*), are vectors called *dual variables* or *Lagrange multiplier vectors*. If the primary variable  $\mathbf{x} \in \mathbb{R}^n$ , the total number of variables in this function will be  $n + m + p$ .

(Usually, I use lowercase bold letters to represent vectors; however, I could not bold  $\lambda$  and  $\nu$  here due to the limitation of writing both in LaTeX and markdown. I note this to avoid confusion for the reader.)

### The Lagrange Dual Function

The Lagrange dual function of the optimization problem (or simply the *dual function*) (3.46) is a function of the dual variables, defined as the infimum over  $\mathbf{x}$  of the *Lagrangian*:

$$\begin{aligned} g(\lambda, \nu) &= \inf_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\mathbf{x}, \lambda, \nu) \\ &= \inf_{\mathbf{x} \in \mathcal{D}} \left( f_0(\mathbf{x}) + \sum_{i=1}^m \lambda_i f_i(\mathbf{x}) + \sum_{j=1}^p \nu_j h_j(\mathbf{x}) \right) \end{aligned}$$

If the *Lagrangian* is not bounded below, the dual function at  $\lambda, \nu$  will take the value  $-\infty$ .

#### Notes:

- The inf is taken over the domain  $\mathbf{x} \in \mathcal{D}$ , which is the domain of the problem (the intersection of the domains of all functions in the problem). This domain is different from the *feasible set*. Usually, the *feasible set* is a subset of the domain  $\mathcal{D}$ .
- For each  $\mathbf{x}$ , the *Lagrangian* is an *affine* function of  $(\lambda, \nu)$ , which is a *concave function*. Therefore, the *dual function* is the *pointwise infimum* of (potentially infinitely many) concave functions, which is a concave function. Thus, *the dual function of any optimization problem is concave, regardless of whether the original problem is convex*. Recall that the *pointwise supremum* of convex functions is a convex function, and a function is *concave* if negating it results in a *convex* function.

### Lower Bound on the Optimal Value

If  $p^*$  is the *optimal value* of problem (3.46), then for any dual variables  $\lambda_i \geq 0, \forall i$ , and  $\nu$ , we have:

$$g(\lambda, \nu) \leq p^* \tag{3.47}$$

This property can be easily proven. Suppose  $\mathbf{x}_0$  is any *feasible* point of problem (3.46), meaning it satisfies the constraints  $f_i(\mathbf{x}_0) \leq 0, \forall i = 1, \dots, m; h_j(\mathbf{x}_0) = 0, \forall j = 1, \dots, p$ . We then have:

$$\sum_{i=1}^m \lambda_i f_i(\mathbf{x}_0) + \sum_{j=1}^p \nu_j h_j(\mathbf{x}_0) \leq 0 \Rightarrow \mathcal{L}(\mathbf{x}_0, \lambda, \nu) \leq f_0(\mathbf{x}_0)$$

Since this is true for all feasible  $\mathbf{x}_0$ , we have the following important property:

$$g(\lambda, \nu) = \inf_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\mathbf{x}, \lambda, \nu) \leq \mathcal{L}(\mathbf{x}_0, \lambda, \nu) \leq f_0(\mathbf{x}_0)$$

When  $\mathbf{x}_0 = \mathbf{x}^*$ , we get inequality (3.47).

### 3.16.3 The Lagrange Dual Problem

For each pair  $(\lambda, \nu)$ , the Lagrange dual function provides a lower bound for the *optimal value*  $p^*$  of the original problem (3.46). The question is: for which pair  $(\lambda, \nu)$  do we get the best lower bound for  $p^*$ ? In other words, we need to solve the problem:

$$\begin{aligned} \lambda^*, \nu^* &= \arg \max_{\lambda, \nu} g(\lambda, \nu) \\ \text{subject to: } &\lambda \succeq 0 \end{aligned} \tag{3.48}$$

**One important point:** since  $g(\lambda, \nu)$  is *concave* and the constraint functions  $f_i(\lambda) = -\lambda_i$  are *convex* functions, problem (3.48) is a convex optimization problem. Therefore, in many cases, the solution can be easier to find than for the original problem. Note that the dual problem (3.48) is convex regardless of whether the original problem (3.46) is convex.

This problem is called the *Lagrange dual problem* associated with problem (3.46). Problem (3.46) is also referred to as the *primal problem*. Additionally, there is a concept called *dual feasible*, which refers to the *feasible set* of the dual problem, including the condition  $\lambda \succeq 0$  and the hidden condition  $g(\lambda, \nu) > -\infty$  (since we are looking for the maximum value of the function,  $g(\lambda, \nu) = -\infty$  is obviously not interesting).

The solution of problem (3.48), denoted by  $\lambda^*, \nu^*$ , is called *dual optimal* or *optimal Lagrange multipliers*.

Note that the hidden condition  $g(\lambda, \nu) > -\infty$  can be explicitly written in many cases. Returning to the example above, the hidden condition can be written as  $\mathbf{c} + \mathbf{A}^T \nu - \lambda = 0$ . This is an affine function. Therefore, even with this additional constraint, we still have a convex problem.

### Weak Duality

Let the optimal value of the dual problem (3.48) be denoted by  $d^*$ . According to (3.48), we know that:

$$d^* \leq p^*$$

even if the original problem is not convex.

This simple property is called *weak duality*. Though simple, it is extremely important.

**From here we observe two things:**

- If the primal problem is not bounded below, i.e.,  $p^* = -\infty$ , then we must have  $d^* = -\infty$ , meaning the Lagrange dual problem is *infeasible* (i.e., there is no value that satisfies the constraints).
- If the dual problem is not bounded above, i.e.,  $d^* = +\infty$ , we must have  $p^* = +\infty$ , meaning the original problem is *infeasible*.

The value  $p^* - d^*$  is called the *optimal duality gap*. This gap is always non-negative.

Sometimes, there are problems (convex or non-convex) that are very difficult to solve, but at least if we can find  $d^*$ , we can determine a lower bound for the original problem. Finding  $d^*$  is often feasible because the dual problem is always convex.

### Strong Duality and Slater's Constraint Qualification

If the equality  $p^* = d^*$  holds, the *optimal duality gap* is zero, and we say that *strong duality* occurs. In this case, solving the dual problem allows us to find the exact optimal value of the primal problem.

Unfortunately, *strong duality* does not always occur in optimization problems. However, if the primal problem is convex, i.e., it has the form:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } &f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ &\mathbf{Ax} = \mathbf{b} \end{aligned} \tag{3.49}$$

where  $f_0, f_1, \dots, f_m$  are convex functions, we *often* (but not always) have *strong duality*. There has been much research establishing conditions beyond convexity for *strong duality* to occur. These conditions are often called *constraint qualifications*.

One of the simplest *constraint qualifications* is *Slater's condition*.

**Definition:** A *feasible* point of problem (3.49) is called *strictly feasible* if:

$$f_i(\mathbf{x}) < 0, \quad i = 1, 2, \dots, m, \quad \mathbf{Ax} = \mathbf{b}$$

**Slater's Theorem:** If there exists a *strictly feasible* point (and the primal problem is convex), then *strong duality* holds.

This fairly simple condition will be helpful for many subsequent optimization problems.

Note:

- *Strong duality* does not always occur. For convex problems, it occurs more frequently. There are convex problems for which *strong duality* does not hold.
- There are non-convex problems for which *strong duality* still holds.

### 3.16.4 Optimality Conditions

#### Complementary Slackness

Assume that *strong duality* holds. Let  $\mathbf{x}^*$  be an *optimal* point of the primal problem, and let  $(\lambda^*, \nu^*)$  be the *optimal* solution of the dual problem. We have:

$$\begin{aligned} f_0(\mathbf{x}^*) &= g(\lambda^*, \nu^*) \\ &= \inf_{\mathbf{x}} \left( f_0(\mathbf{x}) + \sum_{i=1}^m \lambda_i^* f_i(\mathbf{x}) + \sum_{j=1}^p \nu_j^* h_j(\mathbf{x}) \right) \\ &\leq f_0(\mathbf{x}^*) + \sum_{i=1}^m \lambda_i^* f_i(\mathbf{x}^*) + \sum_{j=1}^p \nu_j^* h_j(\mathbf{x}^*) \\ &\leq f_0(\mathbf{x}^*) \end{aligned}$$

- The first line is due to *strong duality*.
- The second line is due to the definition of the dual function.
- The third line is obvious because the infimum of a function is less than or equal to the value of that function at any other point.
- The fourth line holds because of the constraints  $f_i(\mathbf{x}^*) \leq 0, \lambda_i \geq 0, i = 1, 2, \dots, m$  and  $h_j(\mathbf{x}^*) = 0$ .

From here, we can see that the equality in the third and fourth lines must simultaneously hold. This leads to two additional interesting observations:

- $\mathbf{x}^*$  is an *optimal* point of  $g(\lambda^*, \nu^*)$ .
- More interestingly:  

$$\sum_{i=1}^m \lambda_i^* f_i(\mathbf{x}^*) = 0$$

Since each term in the above sum is non-positive due to  $\lambda_i^* \geq 0, f_i \leq 0$ , we conclude that:

$$\lambda_i^* f_i(\mathbf{x}^*) = 0, \quad i = 1, 2, \dots, m$$

This condition is called *complementary slackness*. From this, we can infer:

$$\lambda_i^* > 0 \Rightarrow f_i(\mathbf{x}^*) = 0 \quad (3.50)$$

$$f_i(\mathbf{x}^*) < 0 \Rightarrow \lambda_i^* = 0 \quad (3.51)$$

That is, one of the two values must always be zero.

### KKT Optimality Conditions

We still assume that the functions under consideration are differentiable, and the optimization problem does not necessarily have to be convex.

### KKT Conditions for Non-Convex Problems

Assume that *strong duality* holds. Let  $\mathbf{x}^*$  and  $(\lambda^*, \nu^*)$  be *any primal and dual optimal points*. Since  $\mathbf{x}^*$  optimizes the differentiable function  $\mathcal{L}(\mathbf{x}, \lambda^*, \nu^*)$ , the derivative of the Lagrangian at  $\mathbf{x}^*$  must be zero.

The Karush-Kuhn-Tucker (KKT) conditions state that  $\mathbf{x}^*, \lambda^*, \nu^*$  must satisfy:

$$\begin{aligned} f_i(\mathbf{x}^*) &\leq 0, \quad i = 1, 2, \dots, m \\ h_j(\mathbf{x}^*) &= 0, \quad j = 1, 2, \dots, p \\ \lambda_i^* &\geq 0, \quad i = 1, 2, \dots, m \\ \lambda_i^* f_i(\mathbf{x}^*) &= 0, \quad i = 1, 2, \dots, m \\ \nabla f_0(\mathbf{x}^*) + \sum_{i=1}^m \lambda_i^* \nabla f_i(\mathbf{x}^*) + \sum_{j=1}^p \nu_j^* \nabla h_j(\mathbf{x}^*) &= 0 \end{aligned}$$

These are the *necessary conditions* for  $\mathbf{x}^*, \lambda^*, \nu^*$  to be solutions of both problems.

### KKT Conditions for Convex Problems

For convex problems where *strong duality* holds, the KKT conditions above are also *sufficient conditions*. Therefore, for convex problems with differentiable objective and constraint functions, any point that satisfies the KKT conditions is *primal and dual optimal* for the primal and dual problems.

From this, we can conclude that: For a convex problem with Slater's condition satisfied (implying *strong duality*), the KKT conditions are necessary and sufficient for optimality.

The KKT conditions are very important in optimization. In some special cases (as we will see in the upcoming Support Vector Machine lesson), solving the system of (in)equations given by the KKT conditions is feasible. Many optimization algorithms are based on solving the system of KKT conditions.

### Example: Equality Constrained Convex Quadratic Minimization

Consider the problem:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} \frac{1}{2} \mathbf{x}^T \mathbf{P} \mathbf{x} + \mathbf{q}^T \mathbf{x} + r \\ \text{subject to:} \quad \mathbf{A} \mathbf{x} &= \mathbf{b} \end{aligned}$$

where  $\mathbf{P} \in \mathbb{S}_+^n$  (the set of symmetric positive semi-definite matrices).

Lagrangian:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\nu}) = \frac{1}{2} \mathbf{x}^T \mathbf{P} \mathbf{x} + \mathbf{q}^T \mathbf{x} + r + \boldsymbol{\nu}^T (\mathbf{A} \mathbf{x} - \mathbf{b})$$

The KKT conditions for this problem are:

$$\begin{aligned}\mathbf{A} \mathbf{x}^* &= \mathbf{b} \\ \mathbf{P} \mathbf{x}^* + \mathbf{q} + \mathbf{A}^T \boldsymbol{\nu}^* &= 0\end{aligned}$$

The second equation is the equation of the Lagrangian derivative at  $\mathbf{x}^*$  being zero.

This system can be rewritten simply as a simple linear equation:  $\begin{bmatrix} \mathbf{P} & \mathbf{A}^T \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}^* \\ \boldsymbol{\nu}^* \end{bmatrix} = \begin{bmatrix} -\mathbf{q} \\ \mathbf{b} \end{bmatrix}$

# Chapter 4

## Basic Machine Learning

### 4.1 Dataset

#### 4.1.1 Definition of a Dataset

A **dataset** (or data set) is a collection of data. In the case of *tabular data*, a dataset corresponds to one or more database tables, where each column represents a specific variable, and each row corresponds to a particular record in the dataset. For instance, a dataset might list values for variables such as *height* and *weight* for each entry in the dataset. Datasets can also be collections of documents or files.

In the field of open data, a *dataset* serves as the basic unit of information released in a public open data repository. For example, the European *data.europa.eu*<sup>1</sup> portal aggregates more than a million datasets for public use.

**In statistics** datasets often derive from real-world observations collected by sampling a *statistical population*. Each row typically represents the observations for one member of the population. Datasets can also be generated algorithmically, often for testing and validating statistical software. Classical statistical software, like SPSS, presents data in a traditional dataset structure. To address missing or suspicious data, *imputation methods* may be applied to complete the dataset.

#### 4.1.2 Properties of a Dataset

Several properties characterize a dataset's structure:

- **Attributes or Variables:** These define the dataset's scope and represent measurable characteristics.

---

<sup>1</sup><https://data.europa.eu>

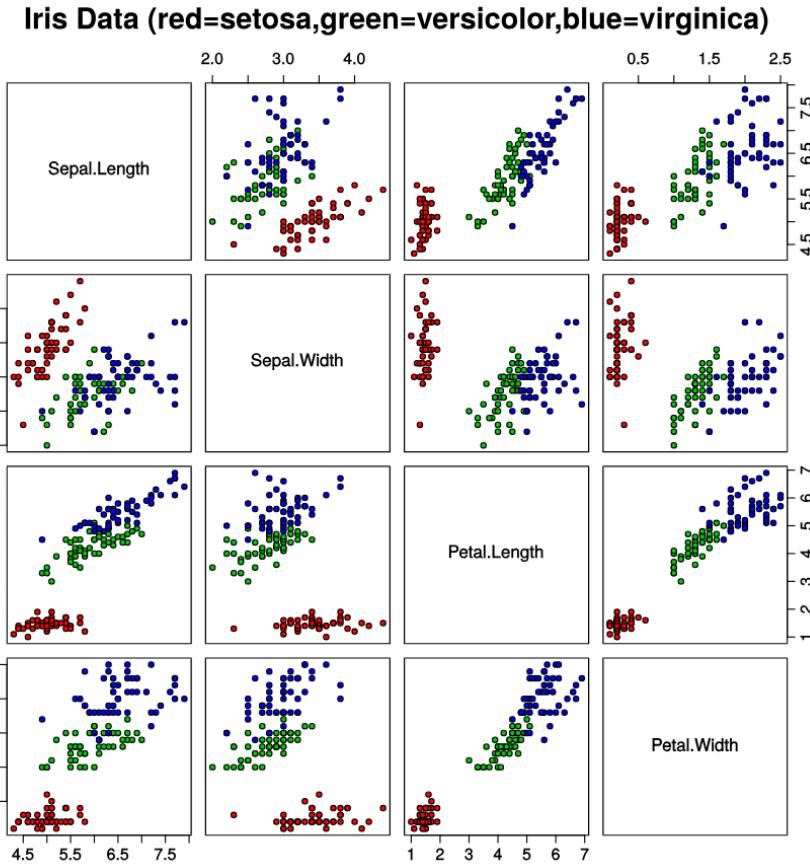


Figure 4.1: Various plots of the multivariate data set Iris flower data set introduced by Ronald Fisher (1936).

- **Data Types:** Values within the dataset can be numerical (e.g., real numbers or integers) or nominal (e.g., categories representing non-numerical data). The type of each variable must be consistent throughout the dataset.
- **Levels of Measurement:** Variables may fall into categories of measurement such as nominal, ordinal, interval, or ratio levels, depending on the nature of the values.
- **Statistical Measures:** Statistical properties like *standard deviation* and *kurtosis* provide insight into the distribution and variability of the data.
- **Missing Values:** Missing data, often indicated with specific symbols or codes, may be present and may require methods such as imputation for handling them.

### 4.1.3 Classical Datasets in Statistical Literature

Several classical datasets are frequently referenced in statistical literature:

- **Iris Flower Dataset:** A multivariate dataset introduced by Ronald Fisher (1936), available from the University of California-Irvine Machine Learning Repository.

- **MNIST Database:** Consisting of images of handwritten digits, commonly used to test classification and clustering algorithms.
- **Categorical Data Analysis Datasets:** Available through UCLA Advanced Research Computing, these datasets accompany the book, *An Introduction to Categorical Data Analysis*.
- **Robust Statistics Datasets:** Used in *Robust Regression and Outlier Detection* by Rousseeuw and Leroy (1968), accessible via the University of Cologne.
- **Time Series Data:** Datasets accompanying Chatfield's book, *The Analysis of Time Series*, are hosted by StatLib.
- **Extreme Values:** Stuart Coles' book, *An Introduction to the Statistical Modeling of Extreme Values*, includes these datasets.
- **Bayesian Data Analysis:** Datasets for the book by Andrew Gelman are archived and available online.
- **Bupa Liver Dataset:** Frequently referenced in machine learning literature.
- **Anscombe's Quartet:** A small dataset illustrating the importance of graphing data to avoid statistical fallacies.

#### 4.1.4 Imbalanced Datasets

An **imbalanced dataset** occurs when the classes in the dataset are not represented equally. In a binary classification problem, if one class significantly outnumbers the other, we refer to the majority class (more instances) and the minority class (fewer instances). Imbalanced datasets can lead to biased models that favor the majority class, thus compromising the model's ability to generalize.

The following table provides generally accepted names and ranges for different degrees of imbalance:

Percentage of data belonging to minority class	Degree of imbalance
20–40% of the dataset	Mild
1–20% of the dataset	Moderate
<1% of the dataset	Extreme

Table 4.1: Degree of imbalance based on percentage of data belonging to the minority class.

For example, consider a dataset in an AI-generated text detection project where the minority class (human-written text) represents only 0.5% of the dataset, while the majority class (AI-generated text) represents 99.5%. Extremely imbalanced datasets like this are common in text analysis.

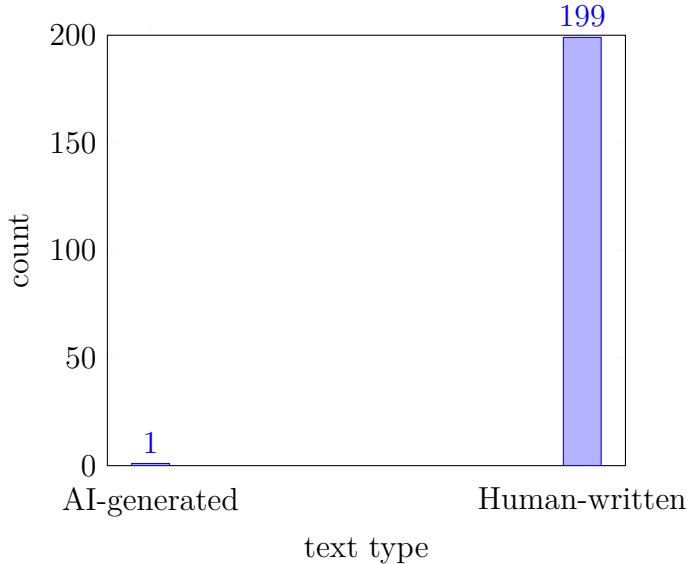


Figure 4.2: Extremely imbalanced dataset in AI-generated text detection.

Imbalanced datasets can cause issues such as:

- Models biased toward the majority class, resulting in poor performance on the minority class.
- Reduced accuracy in detecting the minority class, often critical in fields like fraud detection and medical diagnosis.

**Techniques to Address Imbalanced Datasets** There are several techniques to handle imbalanced datasets effectively:

- **Resampling Methods:**
  - **Oversampling** the minority class by duplicating its examples to balance the dataset.
  - **Downsampling** the majority class by training on a reduced subset of majority examples. For instance, in a virus detection dataset, downsampling by a factor of 10 changes the ratio from 1 positive to 200 negatives (0.5%) to 1 positive to 20 negatives (5%), improving balance.
- **Upweighting the Downsampled Class:** After downsampling, assign an example weight to the majority class proportional to the downsampling factor, increasing its importance during training. For example, if downsampling by a factor of 10, apply a weight of 10 to the downsampled class examples.
- **Synthetic Data Generation:** Use techniques like *SMOTE (Synthetic Minority Over-sampling Technique)* to generate synthetic examples for the minority class. It addresses

class imbalance by generating synthetic data points for the minority class. It does so by selecting a sample and its nearest neighbors, creating new instances within this neighborhood. However, SMOTE may introduce noise if minority samples overlap with other classes. A variation, Borderline-SMOTE, focuses only on samples near the decision boundary, avoiding overlap. These methods help improve classification in imbalanced datasets.

- **Algorithmic Adjustments:** Modify algorithms to handle imbalanced data more effectively, such as adjusting decision thresholds or applying algorithms that consider class weights.
- **Evaluation Metrics:** Utilize metrics like *precision*, *recall*, *F1-score*, and *AUC-ROC* instead of accuracy to better evaluate performance on imbalanced data.
- **Experiment with Rebalance Ratios:** To determine the optimal downsampling and upweighting ratios, treat them as hyperparameters. Factors such as batch size, imbalance ratio, and training set size should be considered; ideally, each batch should contain multiple examples of the minority class.

## 4.2 Synthetic Minority Over-sampling Technique (SMOTE)

### 4.2.1 Introduction to Class Imbalance

In many real-world datasets, especially in fields such as fraud detection, medical diagnosis, and anomaly detection, there exists a significant issue of **class imbalance**. This occurs when the number of instances in one class (usually the majority or negative class) far exceeds the number of instances in another class (usually the minority or positive class).

#### Example of Class Imbalance

Suppose we are building a classifier to detect AI-generated text. In a dataset of 10,000 text samples, only 100 samples are AI-generated, while the remaining 9,900 samples are human-written. This results in a highly imbalanced dataset, with only 1% of the instances belonging to the minority class (AI-generated text).

Class imbalance often leads to biased models that tend to predict the majority class, thereby reducing the effectiveness of classifiers on the minority class. A commonly used technique to address this problem is **SMOTE** (Synthetic Minority Over-sampling Technique).

### 4.2.2 Addressing Class Imbalance: Traditional Over-sampling Techniques

Before diving into the details of **SMOTE** (Synthetic Minority Over-sampling Technique), it's essential to understand traditional methods used to address class imbalance. One commonly used approach is over-sampling, which involves increasing the number of instances in the minority class to balance the dataset. Traditional over-sampling techniques include:

- **Random Over-sampling:** This method involves randomly duplicating instances from the minority class until the dataset is balanced. While it is simple and easy to implement, random over-sampling has a significant downside—it can lead to **overfitting**. The classifier may learn to memorize these duplicated instances rather than generalizing patterns, resulting in poor performance on unseen data.
- **Over-sampling with Replacement:** A variation of random over-sampling, where instances from the minority class are duplicated with replacement. Although this can slightly reduce overfitting, it still does not address the root problem of generating diverse examples for the minority class.

Traditional over-sampling techniques, such as random over-sampling, simply replicate existing examples, which can lead to overfitting and a lack of diversity in the training data. This prevents the model from expanding the decision boundary of the minority class, thus failing to generalize well to new data.

### 4.2.3 What is SMOTE?

To overcome the limitations of traditional over-sampling techniques, **SMOTE** was introduced by Chawla et al. in 2002. Instead of merely duplicating minority class instances, SMOTE generates **new synthetic examples** by interpolating between existing minority class samples. This approach creates more diverse synthetic instances, which helps improve the classifier's ability to generalize to unseen data.

### 4.2.4 How Does SMOTE Work?

The following steps illustrate the SMOTE algorithm:

#### Step 1: Randomly Select a Minority Class Instance

Randomly select a sample  $x_i$  from the minority class.

#### Step 2: Find the $k$ -Nearest Neighbors

Identify the  $k$ -nearest neighbors of  $x_i$  among the other instances in the minority class. Let  $x_{nn}$  be one of these neighbors.

#### Step 3: Generate a Synthetic Sample

A synthetic sample  $x_{new}$  is generated using the formula:

$$x_{new} = x_i + \delta \times (x_{nn} - x_i)$$

where  $\delta$  is a random number between 0 and 1. This ensures that the new synthetic sample lies along the line segment joining  $x_i$  and  $x_{nn}$ .

#### Example of SMOTE

Suppose we have two minority class instances:

$$x_i = (2, 3), \quad x_{nn} = (3, 5)$$

To generate a synthetic sample with  $\delta = 0.6$ :

$$x_{new} = (2, 3) + 0.6 \times ((3, 5) - (2, 3)) = (2, 3) + (0.6, 1.2) = (2.6, 4.2)$$

Thus, the new synthetic instance is  $x_{new} = (2.6, 4.2)$ .

### 4.2.5 Advantages and Limitations of SMOTE

#### Advantages

- **Improves Minority Class Recognition:** By creating synthetic samples, SMOTE reduces the classifier's bias toward the majority class, improving recognition of minority class instances.

- **Prevents Overfitting:** Unlike over-sampling with replacement, SMOTE does not simply duplicate minority class examples, thereby reducing the risk of overfitting.

## Limitations

- **Risk of Overlapping Classes:** SMOTE may generate synthetic samples that overlap with the majority class, especially when the classes are not well-separated.
- **Not Effective for High-Dimensional Data:** In high-dimensional feature spaces, the synthetic samples may not be meaningful, potentially leading to poorer model performance.

### 4.2.6 Implementation of SMOTE in Python

In this section, we demonstrate how to handle class imbalance using the Synthetic Minority Over-sampling Technique (SMOTE). Below is an example using the `imblearn` library in Python to generate a synthetic imbalanced dataset with three classes, and then applying SMOTE to balance the dataset.

Listing 4.1: SMOTE Implementation on Multi-Class Data

```

1 import pandas as pd
2 import numpy as np
3 from sklearn.datasets import make_classification
4 from collections import Counter
5 from imblearn.over_sampling import SMOTE
6 import matplotlib.pyplot as plt
7
8
9 # Step 1: Generate a synthetic imbalanced dataset with 3 classes
10 def generate_imbalanced_dataset():
11     X, y = make_classification(
12         n_samples=1000,           # Total samples
13         n_features=2,            # Number of features
14         n_informative=2,          # Number of informative features
15         n_redundant=0,            # Number of redundant features
16         n_clusters_per_class=1,   # Clusters per class
17         n_classes=3,              # Number of classes
18         weights=[0.8, 0.1, 0.1], # Class distribution
19         random_state=42           # For reproducibility
20     )
21     return X, y
22
23 # Step 2: Apply SMOTE with sampling_strategy='not majority'
24 def apply_smote(X, y):
25     print("Class distribution before SMOTE:")
26     print(Counter(y))
27
28     # Apply SMOTE to balance the dataset, except the majority class
29     smote = SMOTE(sampling_strategy='not majority', random_state=42)
30     X_resampled, y_resampled = smote.fit_resample(X, y)

```

```

30
31     print("\nClass distribution after applying SMOTE ('not majority'):")
32     print(Counter(y_resampled))
33
34     return X_resampled, y_resampled
35
36 # Step 3: Visualize the dataset before and after SMOTE
37 def plot_datasets(X_original, y_original, X_resampled, y_resampled):
38     plt.figure(figsize=(12, 5))
39
40     # Before SMOTE
41     plt.subplot(1, 2, 1)
42     plt.title("Original Dataset (Imbalanced)")
43     for class_label in np.unique(y_original):
44         plt.scatter(X_original[y_original == class_label][:, 0],
45                     X_original[y_original == class_label][:, 1],
46                     label=f'Class {class_label}', alpha=0.6)
47     plt.legend()
48
49     # After SMOTE
50     plt.subplot(1, 2, 2)
51     plt.title("Dataset After SMOTE")
52     for class_label in np.unique(y_resampled):
53         plt.scatter(X_resampled[y_resampled == class_label][:, 0],
54                     X_resampled[y_resampled == class_label][:, 1],
55                     label=f'Class {class_label}', alpha=0.6)
56     plt.legend()
57
58     plt.show()
59
60 # Main function to run the program
61 def main():
62     # Generate an imbalanced dataset
63     X, y = generate_imbalanced_dataset()
64
65     # Apply SMOTE to balance the dataset
66     X_resampled, y_resampled = apply_smote(X, y)
67
68     # Visualize the datasets before and after SMOTE
69     plot_datasets(X, y, X_resampled, y_resampled)
70
71 # Run the main function
72 if __name__ == "__main__":
73     main()

```

## Class Distribution Before and After SMOTE

The following plot shows the distribution of classes before and after applying SMOTE with the strategy of balancing only the minority classes while keeping the majority class unchanged.

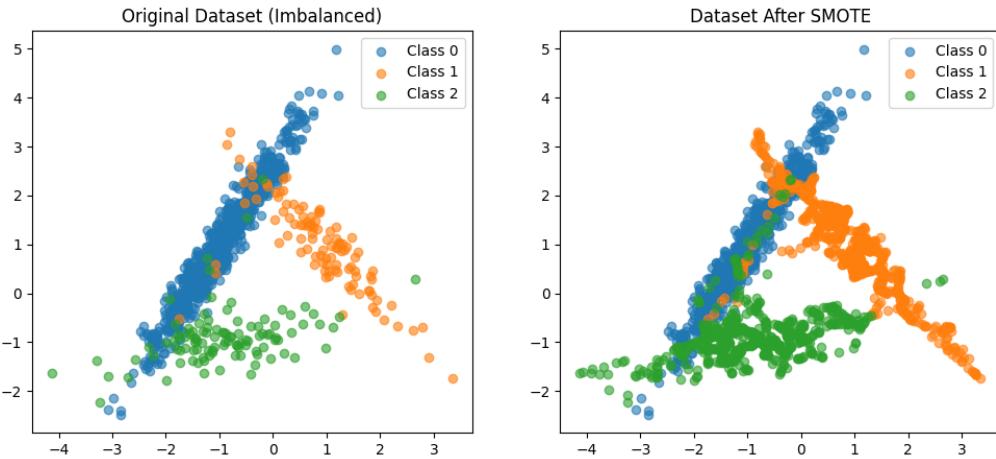


Figure 4.3: Class Distribution Before and After Applying SMOTE

## Explanation

- In the left plot, the dataset is highly imbalanced, with one majority class and two minority classes.
- In the right plot, SMOTE has been applied with the strategy '`not majority`', which balances the minority classes while keeping the majority class unchanged.
- This technique helps improve classifier performance by providing a more balanced dataset without oversampling the majority class.

### 4.2.7 Extensions of SMOTE

While SMOTE is highly effective for addressing class imbalance, it may not always be suitable for all data types or situations. To handle different challenges, several extensions of SMOTE have been developed:

- **ADASYN (Adaptive Synthetic Sampling)**: ADASYN is designed to focus on generating synthetic samples *in regions with high imbalance density*. It adapts to the local density of minority class examples, generating more synthetic samples where the imbalance is severe and fewer where it is not. This helps models learn harder-to-classify regions and improves overall performance.
- **Borderline SMOTE**: Borderline SMOTE focuses on *generating synthetic samples near the decision boundary where the risk of misclassification is highest*. It aims to enhance the classification accuracy by strengthening the presence of minority class samples near the boundary.

- **SMOTE-ENN (Edited Nearest Neighbors):** SMOTE-ENN combines SMOTE with Edited Nearest Neighbors (ENN) to *remove noisy data after generating synthetic samples*. This combination improves the quality of the dataset by eliminating noisy or mislabeled examples.
- **SMOTE + Tomek Links:** This approach refines the dataset further by combining SMOTE with Tomek Links, which removes overlapping instances between classes. This enhances class separability and reduces noise.
- **SMOTE-NC (Nominal Continuous):** SMOTE-NC is specifically designed for datasets with mixed continuous and nominal (categorical) features. It modifies the SMOTE algorithm to handle categorical features separately while generating synthetic samples for continuous features.

Table 4.2: Comparison of SMOTE Extensions

Extension	Best Use Case	Strengths	When to Use
<b>ADASYN</b>	Datasets with varying class densities	Focuses on harder-to-classify regions	When certain areas of feature space are more imbalanced
<b>Borderline SMOTE</b>	Minority class near decision boundary	Enhances classification near class boundaries	When data points overlap and are prone to misclassification
<b>SMOTE-ENN</b>	Noisy datasets	Combines oversampling with data cleaning	When noise reduction is needed after oversampling
<b>SMOTE + Tomek Links</b>	Reducing class overlap	Removes ambiguous samples	When enhancing class separability is crucial
<b>SMOTE-NC</b>	Mixed data types	Handles categorical and continuous features	For datasets with both nominal and continuous features

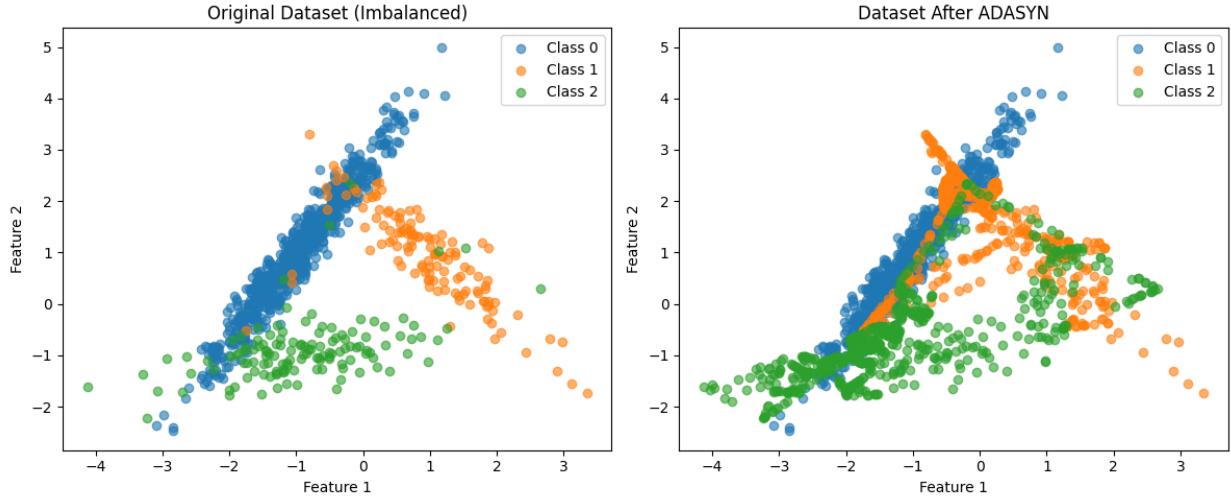


Figure 4.4: Class Distribution Before and After Applying SMOTE Adasyn

#### 4.2.8 Conclusion

Each extension of SMOTE addresses different challenges in handling class imbalance:

- ADASYN adapts to local data density.
- Borderline SMOTE focuses on hard-to-classify regions near decision boundaries.
- SMOTE-ENN and SMOTE-Tomek combine over-sampling with noise reduction techniques.
- SMOTE-NC is suitable for datasets with mixed feature types.

These extensions provide flexibility in handling various data scenarios, enhancing model performance on imbalanced datasets.

#### 4.2.9 Applications of SMOTE

SMOTE can be particularly useful in applications such as:

- **Fraud Detection:** Balancing fraud detection datasets to improve the identification of fraudulent transactions.
- **Medical Diagnosis:** Enhancing the detection of rare diseases in healthcare datasets.
- **Information Retrieval (IR):** Addressing class imbalance in text categorization by generating synthetic documents for under-represented categories.

## 4.3 Bayesian Modeling and Inference

### 4.3.1 Bayesian Models

Bayesian models, computational or otherwise, have two defining characteristics:

- Unknown quantities are described using probability distributions. We call these quantities *parameters*.
- Bayes' theorem is used to update the values of the parameters conditioned on the data. We can also see this process as a reallocation of probabilities.

At a high level, we can describe the process of constructing Bayesian modeling in 3 steps.

1. Given some data and some assumptions on how this data could have been generated, we design a model by combining and transforming random variables.
2. We use Bayes' theorem to condition our models to the available data. We call this process **inference**, and as a result, we obtain a posterior distribution. We hope the data reduces the uncertainty for possible parameter values, though this is not a guarantee of any Bayesian model.
3. We criticize the model by checking whether the model makes sense according to different criteria, including the data and our expertise in domain knowledge. Because we generally are uncertain about the models themselves, we sometimes compare several models.

### 4.3.2 Bayesian Inference

In colloquial terms, inference is associated with obtaining conclusions based on evidence and reasoning. Bayesian inference is a particular form of statistical inference based on combining probability distributions in order to obtain other probability distributions. Bayes' theorem provides us with a general recipe to estimate the value of the parameter  $\theta$  given that we have observed some data  $\mathbf{Y}$ :

$$\underbrace{p(\theta | \mathbf{Y})}_{\text{posterior}} = \frac{\overbrace{p(\mathbf{Y} | \theta)}^{\text{likelihood}} \overbrace{p(\theta)}^{\text{prior}}}{\underbrace{p(\mathbf{Y})}_{\text{marginal likelihood}}} \quad (4.1)$$

**Example** Suppose we are building a model to detect whether a given text is written by a human or generated by an AI. We want to estimate the probability that a text is AI-generated ( $\theta$ ) given the evidence from some observed features of the text, such as average sentence length, frequency of certain keywords, or stylistic patterns ( $\mathbf{Y}$ ).

In Bayesian terms, we are interested in the **posterior probability**  $p(\boldsymbol{\theta} \mid \mathbf{Y})$ , which tells us the probability that the text is AI-generated after observing the data  $\mathbf{Y}$ . According to Bayes' theorem:

$$p(\boldsymbol{\theta} \mid \mathbf{Y}) = \frac{p(\mathbf{Y} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathbf{Y})} \quad (4.2)$$

- **Prior ( $p(\boldsymbol{\theta})$ ):** This represents our initial belief about whether a given text is AI-generated before we observe any features. For instance, if we know that historically, 30% of texts are AI-generated, we set  $p(\boldsymbol{\theta} = \text{AI}) = 0.3$  and  $p(\boldsymbol{\theta} = \text{Human}) = 0.7$ .
- **Likelihood ( $p(\mathbf{Y} \mid \boldsymbol{\theta})$ ):** This represents the probability of observing the features  $\mathbf{Y}$  given that the text is either AI-generated or human-written. For example, suppose we observe that texts generated by AI tend to have an average sentence length of 20 words, while human-written texts typically have an average sentence length of 15 words.
- **Marginal Likelihood ( $p(\mathbf{Y})$ ):** This is the normalizing constant ensuring that the posterior is a valid probability distribution. It is computed by summing (or integrating) over all possible values of  $\boldsymbol{\theta}$ :

$$p(\mathbf{Y}) = p(\mathbf{Y} \mid \text{AI})p(\text{AI}) + p(\mathbf{Y} \mid \text{Human})p(\text{Human}) \quad (4.3)$$

Let's assume the following:

### 1. Prior:

$$p(\text{AI}) = 0.3, \quad p(\text{Human}) = 0.7$$

### 2. Likelihoods based on observed feature $\mathbf{Y}$ (e.g., average sentence length):

$$p(\mathbf{Y} \mid \text{AI}) = 0.8, \quad p(\mathbf{Y} \mid \text{Human}) = 0.4$$

### 3. Marginal Likelihood:

$$p(\mathbf{Y}) = p(\mathbf{Y} \mid \text{AI})p(\text{AI}) + p(\mathbf{Y} \mid \text{Human})p(\text{Human})$$

Substituting the values:

$$p(\mathbf{Y}) = (0.8 \times 0.3) + (0.4 \times 0.7) = 0.24 + 0.28 = 0.52$$

### 4. Posterior Calculation:

Now, we can compute the posterior probability of the text being AI-generated:

$$p(\text{AI} \mid \mathbf{Y}) = \frac{p(\mathbf{Y} \mid \text{AI}) p(\text{AI})}{p(\mathbf{Y})}$$

Substituting the known values:

$$p(\text{AI} \mid \mathbf{Y}) = \frac{0.8 \times 0.3}{0.52} = \frac{0.24}{0.52} \approx 0.46$$

Similarly, the posterior probability that the text is human-written:

$$p(\text{Human} \mid \mathbf{Y}) = \frac{p(\mathbf{Y} \mid \text{Human}) p(\text{Human})}{p(\mathbf{Y})}$$

$$p(\text{Human} \mid \mathbf{Y}) = \frac{0.4 \times 0.7}{0.52} = \frac{0.28}{0.52} \approx 0.54$$

### 4.3.3 Interpretation

After observing the feature  $\mathbf{Y}$ , the posterior probability of the text being AI-generated is approximately 46%, while the probability of it being human-written is 54%. Even though the likelihood favored the AI hypothesis ( $p(\mathbf{Y} \mid \text{AI}) = 0.8$ ), the prior knowledge that only 30% of texts are AI-generated influenced the final posterior, resulting in a higher probability for the human hypothesis.

#### Bayesian Triad

The likelihood function links the observed data with the unknown parameters, while the prior distribution represents the uncertainty about the parameters before observing the data  $\mathbf{Y}$ . By multiplying them, we obtain the posterior distribution, which is the joint distribution over all the parameters in the model (conditioned on the observed data). Figure 4.5 shows an example of an arbitrary prior, likelihood, and the resulting posterior.

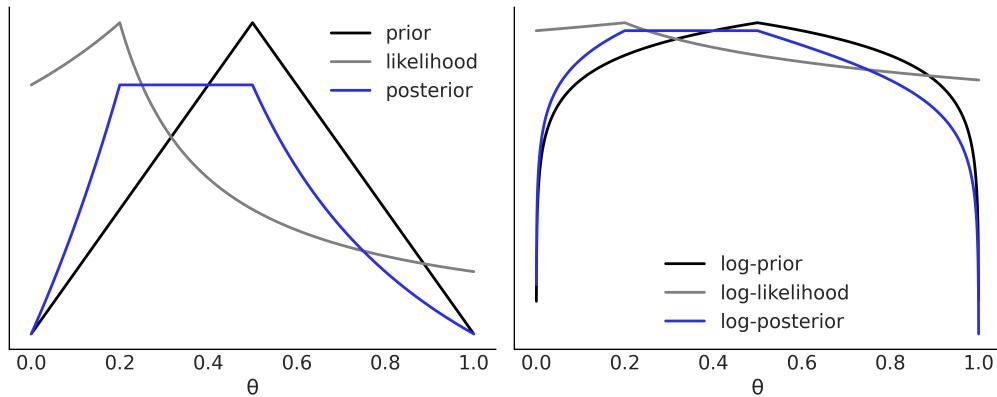


Figure 4.5

On the left panel, a hypothetical prior is seen, indicating that the value  $\theta = 0.5$  is more likely, with the plausibility of the rest of the values decreasing linearly and symmetrically (black). A likelihood showing that the value  $\theta = 0.2$  best agrees with the hypothetical data (gray), and the resulting posterior (blue) represents a compromise between prior and likelihood. The y-axis values are omitted to emphasize that we only care about relative values.

On the right panel, we see the same functions as in the left panel but with the y-axis in log scale. Notice that information about relative values is preserved; for example, the locations of the maxima and minima remain the same in both panels. The log scale is preferred for calculations as computations are numerically more stable.

Notice that while  $\mathbf{Y}$  represents the observed data, it is also a random vector as its values depend on the results of a particular experiment. To obtain a posterior distribution, we regard the data as fixed at the actual observed values. For this reason, a common alternative notation is to use  $y_{obs}$  instead of  $\mathbf{Y}$ .

#### 4.3.4 Evaluating the Posterior

As you can see, evaluating the posterior at each specific *point* is conceptually simple; we just need to multiply a prior by a likelihood. However, that is not enough to inform us about the posterior, as we not only need the posterior probability at that specific *point*, but also in relation to the surrounding *points*. This *global* information of the posterior distribution is represented by the normalizing constant. Unfortunately, difficulties arise from the need to compute the normalizing constant  $p(\mathbf{Y})$ . This is easier to see if we write the marginal likelihood as:

$$p(\mathbf{Y}) = \int_{\Theta} p(\mathbf{Y} | \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta} \quad (4.4)$$

where  $\Theta$  indicates we are integrating over all the possible values of  $\boldsymbol{\theta}$ .

Computing integrals like this can be much harder than it first appears, especially when we realize that for most problems a closed-form expression is not even available. Fortunately, there are numerical methods that can help us with this challenge if used properly. As the marginal likelihood is not generally computed, it is very common to see Bayes' theorem expressed as a proportionality:

$$\underbrace{p(\boldsymbol{\theta} | \mathbf{Y})}_{\text{posterior}} \propto \overbrace{p(\mathbf{Y} | \boldsymbol{\theta})}^{\text{likelihood}} \overbrace{p(\boldsymbol{\theta})}^{\text{prior}} \quad (4.5)$$

One nice feature of Bayesian statistics is that the posterior is (always) a distribution. This fact allows us to make probabilistic statements about the parameters, such as the probability of a parameter  $\tau$  being positive is 0.35, or that the most likely value of  $\phi$  is 12 with a 50% chance of being between 10 and 15. Moreover, we can think of the posterior distribution as the logical consequence of combining a model with the data, ensuring that the probabilistic statements derived from them are mathematically consistent. In the real world, where mathematical purity meets applied mathematics, we must acknowledge that our results are conditioned not only on the data but also on the models. Emphasizing that our inferences are always dependent on the assumptions made by model  $M$ , to clarify the equation above:

$$p(\boldsymbol{\theta} \mid \mathbf{Y}, M) \propto p(\mathbf{Y} \mid \boldsymbol{\theta}, M) p(\boldsymbol{\theta}, M) \quad (4.6)$$

Once we have a posterior distribution, we can use it to derive other quantities of interest. This is generally done by computing expectations, for example:

$$J = \int f(\boldsymbol{\theta}) p(\boldsymbol{\theta} \mid \mathbf{Y}) d\boldsymbol{\theta} \quad (4.7)$$

If  $f$  is the identity function,  $J$  will be the mean:

$$\bar{\boldsymbol{\theta}} = \int_{\Theta} \boldsymbol{\theta} p(\boldsymbol{\theta} \mid \mathbf{Y}) d\boldsymbol{\theta} \quad (4.8)$$

The posterior distribution is the central object in Bayesian statistics, but it is not the only one. Besides making inferences about parameter values, we may also want to make inferences about data. This can be done by computing the **prior predictive distribution**:

$$p(\mathbf{Y}^*) = \int_{\Theta} p(\mathbf{Y}^* \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta} \quad (4.9)$$

This is the expected distribution of the data according to the model (prior and likelihood), i.e., the data we expect given the model before actually observing any data  $\mathbf{Y}^*$ . Notice that Equations 4.4 (marginal likelihood) and 4.9 (prior predictive distribution) are quite similar. The difference is that in the former case, we condition on the observed data  $\mathbf{Y}$ , while in the latter, we do not condition on the observed data. As a result, the marginal likelihood is a number, while the prior predictive distribution is a probability distribution.

We can use samples from the prior predictive distribution as a way to evaluate and calibrate our models using domain knowledge. For example, we may ask questions such as "Is it reasonable for a model of human heights to predict that a human is -1.5 meters tall?" Even before measuring a single person, we can recognize the absurdity of such a prediction. Later in the book, we will see many concrete examples of model evaluation using prior predictive distributions in practice, and how the prior predictive distributions inform the validity, or lack thereof, in subsequent modeling choices.

Another useful quantity to compute is the **posterior predictive distribution**:

$$p(\tilde{\mathbf{Y}} \mid \mathbf{Y}) = \int_{\Theta} p(\tilde{\mathbf{Y}} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid \mathbf{Y}) d\boldsymbol{\theta} \quad (4.10)$$

This is the distribution of expected, future data  $\tilde{\mathbf{Y}}$  according to the posterior  $p(\boldsymbol{\theta} \mid \mathbf{Y})$ , which in turn is a consequence of the model (prior and likelihood) and observed data. In more common terms, this is the data the model is expecting to see after observing the dataset  $\mathbf{Y}$ , i.e., these are the model's predictions. From Equation 4.10, we can see that predictions are computed by integrating out (or marginalizing) over the posterior distribution of parameters. As a result, predictions computed in this way will incorporate the uncertainty about our estimates.

Now that we have a posterior distribution  $p(\boldsymbol{\theta} | \mathbf{Y})$ , we can use it to predict whether future texts are AI-generated. This involves computing the posterior predictive distribution:

$$p(\tilde{\mathbf{Y}} | \mathbf{Y}) = \int_{\Theta} p(\tilde{\mathbf{Y}} | \boldsymbol{\theta}) p(\boldsymbol{\theta} | \mathbf{Y}) d\boldsymbol{\theta} \quad (4.11)$$

## 4.4 Entropy

### 4.4.1 Overview

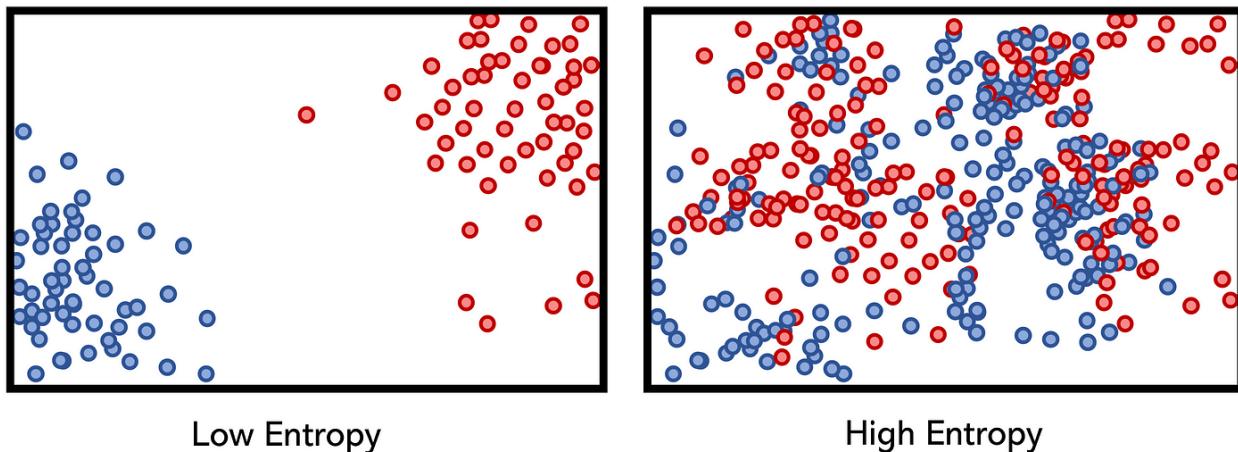


Figure 4.6: Entropy in General

**General** Entropy is a scientific concept associated with randomness, disorder, and uncertainty. Originating in thermodynamics, entropy measures the degree of microscopic randomness in nature and has applications in fields such as *statistical physics* and *information theory*. In these fields, entropy helps quantify uncertainty or disorder in systems, providing insights for disciplines like chemistry, biology, and information systems.

### Entropy in Information Theory

The core idea of **information theory** is that the "informational value" of a communicated message depends on the degree to which the content is surprising. A highly likely event conveys little information, while a highly unlikely event is much more informative. For example, knowing that a specific number will not win a lottery has low informational value, as most numbers will not win. However, knowing a particular number wins the lottery has high informational value due to the rarity of the event.

## 4.4.2 Surprisal and Self-Information

The information content, or *surprisal* (also known as *self-information*), of an event  $E$  is a function that increases as the probability  $p(E)$  decreases. If  $p(E) \approx 1$ , the surprisal is low, but if  $p(E) \approx 0$ , the surprisal is high. This relationship is described by:

$$I(E) = -\log_2(p(E)) \quad \text{or equivalently,} \quad I(E) = \log_2\left(\frac{1}{p(E)}\right),$$

where  $\log$  is the logarithmic function. This formula yields 0 surprise if the event probability is 1, meaning the outcome is fully predictable and carries no information.

## 4.4.3 Entropy as Expected Information

Entropy represents the expected (average) amount of information gained by observing the outcome of a random variable.

For instance, rolling a six-sided die has higher entropy than flipping a fair coin because each outcome in a die roll has a smaller probability ( $p = \frac{1}{6}$ ) than each outcome in a coin toss ( $p = \frac{1}{2}$ ). Entropy is maximized when all outcomes are equally likely, as seen in a fair coin toss with an entropy of 1 bit.

Consider a process with two outcomes: an event with probability  $p$  and its complement with probability  $1 - p$ . The maximum entropy occurs when  $p = 0.5$ , as neither outcome is more expected than the other. Conversely, entropy is zero when  $p = 0$  or  $p = 1$ , as the outcome is predetermined and thus holds no uncertainty or surprise.

In English text, the entropy per character is relatively low due to its predictability. Common letters like 'e' and frequent patterns (e.g., 'th' or 'qu') reduce uncertainty, making the language more compressible. English text entropy typically ranges between 0.6 and 1.3 bits per character, meaning that significant portions of the text can be predicted based on prior knowledge.

## 4.4.4 Shannon Entropy

Named after Boltzmann's H-theorem, Shannon defined the entropy  $H(X)$  of a discrete random variable  $X$ , which takes values in the set  $\mathcal{X}$  with a probability distribution  $p : \mathcal{X} \rightarrow [0, 1]$ . The entropy  $H(X)$  can be defined as the expected value of the information content  $I(X)$ , given by:

$$H(X) = \mathbb{E}[I(X)] = \mathbb{E}[-\log p(X)]$$

where  $\mathbb{E}$  denotes the expected value operator and  $I(X)$  represents the information content of  $X$ .

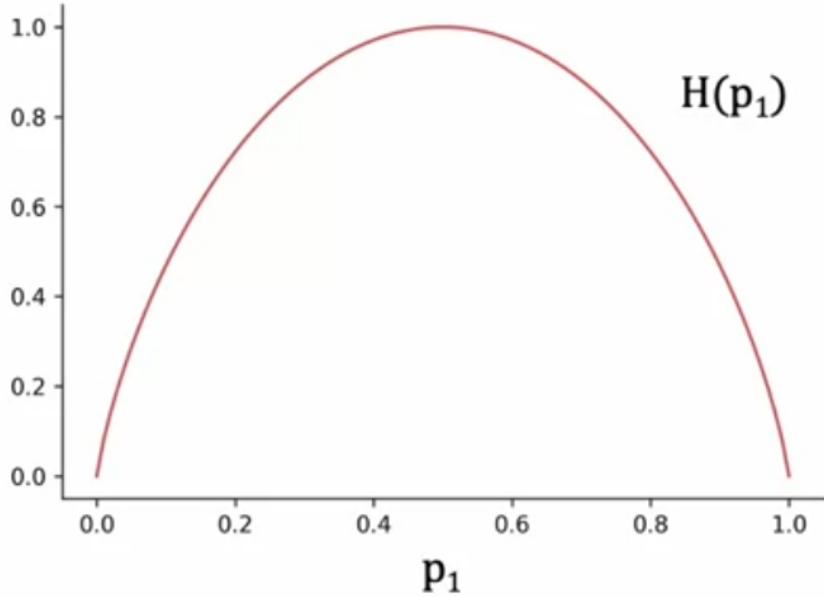


Figure 4.7: Entropy

The entropy formula can be explicitly written as:

$$H(X) = - \sum_{x \in \mathcal{X}} p(x) \log_b p(x),$$

where  $b$  is the base of the logarithm. Common choices of  $b$  are 2 (giving entropy in *bits*), Euler's number  $e$  (for *nats*), and 10 (for *bans*). The entropy value is non-negative, as the negative sign compensates for the fact that logarithms of probabilities (values between 0 and 1) are negative.

In cases where  $p(x) = 0$  for some  $x \in \mathcal{X}$ , the summand  $0 \log_b(0)$  is defined as zero, following the limit:

$$\lim_{p \rightarrow 0^+} p \log_b(p) = 0.$$

#### 4.4.5 Types of Entropy in Information Theory

**Joint Entropy:** For two discrete random variables  $X$  and  $Y$ , the *joint entropy*  $H(X, Y)$  measures the uncertainty of their combined occurrences:

$$H(X, Y) = - \sum_{x,y} p(x, y) \log p(x, y)$$

Joint entropy captures the uncertainty associated with the joint state of  $X$  and  $Y$ .

**Conditional Entropy:** Conditional entropy  $H(X|Y)$ , also known as *equivocation*, measures the uncertainty of  $X$  given that  $Y$  is known:

$$H(X|Y) = - \sum_y p(y) \sum_x p(x|y) \log p(x|y)$$

This quantifies the remaining uncertainty about  $X$  after observing  $Y$ .

### 4.4.6 Mutual Information

**Definition:** Mutual information  $I(X; Y)$  quantifies the amount of information gained about one random variable by observing another. For variables  $X$  and  $Y$ , it is defined as:

$$I(X; Y) = \sum_{x,y} p(x, y) \log \frac{p(x, y)}{p(x)p(y)}$$

It is also expressible as the reduction in uncertainty:

$$I(X; Y) = H(X) - H(X|Y)$$

This shows how knowing  $Y$  reduces the uncertainty of  $X$ .

### 4.4.7 KL Divergence

#### 4.4.7.1 The Entropy of Our Distribution

KL Divergence has its origins in information theory. The primary goal of information theory is to quantify how much information is in data. The most important metric in information theory is called **Entropy**, typically denoted as  $H$ . The definition of Entropy for a probability distribution is:

$$H = - \sum_{i=1}^N p(x_i) \cdot \log p(x_i)$$

If we use  $\log_2$  for our calculation, we can interpret entropy as "the minimum number of bits it would take us to encode our information". In this case, the information would be each observation of teeth counts given our empirical distribution. Given the data that we have observed, our probability distribution has an entropy of 3.12 bits. The number of bits tells us the lower bound for how many bits we would need, on average, to encode the number of teeth we would observe in a single case.

What entropy doesn't tell us is the optimal encoding scheme to help us achieve this compression. Optimal encoding of information is a very interesting topic, but not necessary for understanding KL divergence. The key thing with Entropy is that, simply knowing the theoretical lower bound on the number of bits we need, we have a way to quantify exactly how much information is in our data. Now that we can quantify this, we want to quantify how much information is lost when we substitute our observed distribution for a parameterized approximation.

#### 4.4.7.2 Measuring Information Lost using Kullback-Leibler Divergence

Kullback-Leibler Divergence is just a slight modification of our formula for entropy. Rather than just having our probability distribution  $p$ , we add in our approximating distribution  $q$ . Then we look at the difference of the log values for each:

$$D_{KL}(p\|q) = \sum_{i=1}^N p(x_i) \cdot (\log p(x_i) - \log q(x_i))$$

Essentially, what we're looking at with the KL divergence is the **expectation** of the log difference between the probability of data in the original distribution with the approximating distribution. Again, if we think in terms of  $\log_2$ , we can interpret this as "how many bits of information we expect to lose". We could rewrite our formula in terms of expectation:

$$D_{KL}(p\|q) = E[\log p(x) - \log q(x)]$$

The more common way to see KL divergence written is as follows:

$$D_{KL}(p\|q) = \sum_{i=1}^N p(x_i) \cdot \log \frac{p(x_i)}{q(x_i)}$$

since  $\log a - \log b = \log \frac{a}{b}$ .

With KL divergence, we can calculate exactly how much information is lost when we approximate one distribution with another.

#### Example

##### Basic KL Divergence Between Two Word Distributions

Let's start with a simple example. Suppose we have two word probability distributions — one is the true distribution and the other is generated by our model. We want to compute the KL Divergence to see how far off our model is.

```
import torch
import torch.nn.functional as F

# Define two probability distributions (word probabilities in NLP)
p = torch.tensor([0.1, 0.2, 0.7]) # True distribution
q = torch.tensor([0.3, 0.4, 0.3]) # Predicted distribution from the model

# Compute KL Divergence
kl_div = F.kl_div(torch.log(q), p, reduction='batchmean')
print(f"KL Divergence: {kl_div.item():.4f}")
```

Here, we're calculating the KL Divergence between two distributions:  $p$  represents the real-world word probabilities, and  $q$  is our model's prediction. Notice how we take the  $\log$  of  $q$ , be-

cause `torch.nn.functional.kl_div` expects `q` as log-probabilities. The `reduction='batchmean'` calculates the average over the batch.

Output:

```
KL Divergence: 0.1966
```

This means our model is moderately far off from reality — there's room for improvement.

## KL Divergence Across a Batch of Sentences

When training NLP models, you often work with batches of sentences rather than single distributions. Let's extend our example to compute KL Divergence over a batch of word distributions.

```
p_batch = torch.tensor([[0.1, 0.2, 0.7], [0.3, 0.3, 0.4]]) # True
    distribution batch
q_batch = torch.tensor([[0.3, 0.4, 0.3], [0.4, 0.4, 0.2]]) # Predicted
    distribution batch

# Calculate batch-wise KL Divergence
kl_div_batch = F.kl_div(torch.log(q_batch), p_batch, reduction='batchmean',
    )

print(f"Batch-wise KL Divergence: {kl_div_batch.item():.4f}")
```

In this case, we are computing KL Divergence across multiple sentences, giving us a more holistic view of how well the model is performing over a set of predictions.

## Visualizing KL Divergence

Listing 4.2: Calculating KL Divergence in NLP with Python

```
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn.functional as F

# Generate distributions
x = np.linspace(0.01, 0.99, 100)
p = np.array([0.5, 0.5]) # True distribution (balanced)

# KL Divergence function
def kl_divergence(p, q):
    return np.sum(p * np.log(p / q))

# Calculate KL Divergence for varying q
kl_values = [kl_divergence(p, [q, 1-q]) for q in x]

# Plot KL Divergence curve
plt.plot(x, kl_values)
plt.title('KL Divergence Between Two Word Distributions')
plt.xlabel('q (Predicted Probability)')
plt.ylabel('KL Divergence')
plt.show()
```

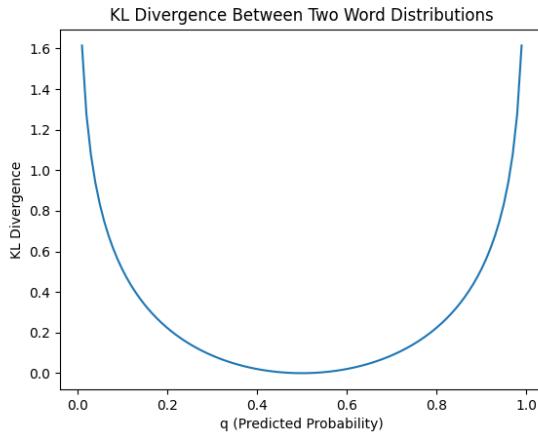


Figure 4.8: Caption

This graph shows how KL Divergence grows as the predicted distribution Q diverges from the true distribution P. If the model's predictions are closer to the true word distribution, KL Divergence is minimized.

In language modeling, we can measure the KL Divergence between the predicted word distribution and the actual word distribution from the corpus to improve the model's predictions.

```
# Simulated word distributions from a language model
true_word_probs = torch.tensor([0.1, 0.6, 0.3]) # Actual word
# distribution
predicted_word_probs = torch.tensor([0.3, 0.5, 0.2]) # Model's prediction

# Calculate KL Divergence
kl_div = F.kl_div(torch.log(predicted_word_probs), true_word_probs,
    reduction='batchmean')
print(f"KL Divergence for Language Model: {kl_div.item():.4f}")
```

The lower the KL Divergence, the better our model is at predicting the next word in the sequence.

Challenges of Using KL Divergence in NLP KL Divergence is incredibly useful but not without its challenges. Here are some common issues:

**Asymmetry:** KL Divergence is not symmetric, which can be problematic when P and Q are far apart.  $D_{KL}(P\|Q) \neq D_{KL}(Q\|P)$ , meaning that swapping distributions can yield drastically different results. **Undefined for Zero Probabilities:** If  $Q(x) = 0$ , for some x where  $P(x) \neq 0$ , the KL Divergence becomes infinite. This can be troublesome when working with sparse or skewed distributions, which are common in NLP tasks. For these reasons, other metrics like Jensen-Shannon Divergence (which is symmetric and smooth) are sometimes preferred in certain NLP scenarios.

## 4.5 Logistic Regression

Symbol	Definition
$x$	training example feature values
$y$	training example targets
$w$	weight parameter
$b$	bias parameter
$m$	number of training examples in the data set
$n$	number of features
$i$	the $i^{th}$ training example in the data set
$j$	the $j^{th}$ feature
$x_j^{(i)}$	value of the $j^{th}$ feature in the $i^{th}$ training example
$w_j^{(i)}$	weight of the $j^{th}$ feature
$f_{\vec{w}, b}(\vec{x}^{(i)})$	model's result of the $i^{th}$ training example
$\hat{y}^{(i)}$	model's prediction of the $i^{th}$ training example
$y^{(i)}$	target of the $i^{th}$ training example
$z$	decision boundary
$g(\vec{w} \cdot \vec{x} + b)$	logistic function
$\mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)})$	loss function
$\mathcal{J}(\vec{w}, b)$	cost function
$\alpha$	learning rate

### 4.5.1 Logistic function

Logistic function is one of sigmoid functions which is commonly applied in logistic regression. We want prediction  $\hat{y}$  between 0 and 1 then logistic functions are appropriate to apply.

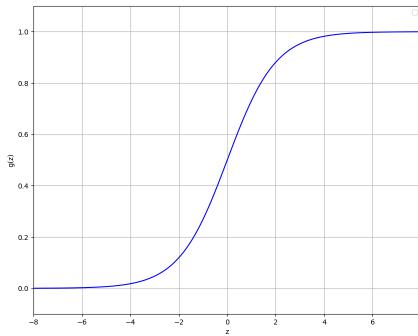


Figure 4.9: Logistic function

$$g(z) = \frac{1}{1 + e^{-z}} \quad (0 < g(z) < 1)$$

$$z = \vec{w} \cdot \vec{x} + b$$

$$f_{\vec{w}, b}(\vec{x}) = g(\vec{w} \cdot \vec{x} + b) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x} + b)}}$$

#### 4.5.2 Decision boundary

$$P(y = 1|x; \vec{w}, b) \text{ or } \hat{y} = 1$$

when  $f_{\vec{w}, b}(\vec{x}) \geq 0.5$  (threshold)

$$g(z) \geq 0.5$$

$$z \geq 0$$

In this case, we say that decision boundary is  $z = 0$ .

$z \geq 0$	$z < 0$
$\vec{w} \cdot \vec{x} + b \geq 0$	$\vec{w} \cdot \vec{x} + b < 0$
$\hat{y} = 1$	$\hat{y} = 0$

Note: Threshold no need to be 0.5. For example, tumor positive(1) malignant, negative(0) be nign, we tend to get a low threshold such as 0.2 or 0.1 for a tumor detection algorithm because we would not like to miss a potential tumor. In contrast, in our project AI-generated text detector with positive(1) AI-generated and negative(0) human-written, we tend to get a high threshold as the consequence of accusing someone of cheating with AI-generated text is more severe than missing. Then, decision boundary =  $z = -\log(\frac{1}{\text{threshold}} - 1)$

#### 4.5.3 Cross-Entropy Loss function

Loss  $\mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)})$  is a measure of the difference of a single example to its target value.  
if  $y^{(i)} = 1$  : We want a loss function that when  $f_{\vec{w}, b}(\vec{x}^{(i)})$  reach 1, the loss reach 0 and when  $f_{\vec{w}, b}(\vec{x}^{(i)})$  reach 0, the loss become higher. Then,  $-\log(f_{\vec{w}, b}(\vec{x}^{(i)}))$  is suitable.

$$\mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)}) = -\log(f_{\vec{w}, b}(\vec{x}^{(i)}))$$

if  $y^{(i)} = 0$  : We want a loss function that when  $f_{\vec{w}, b}(\vec{x}^{(i)})$  reach 0, the loss reach 0 and when  $f_{\vec{w}, b}(\vec{x}^{(i)})$  reach 1, the loss become higher. Then,  $-\log(1 - f_{\vec{w}, b}(\vec{x}^{(i)}))$  is suitable.

$$\mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)}) = -\log(1 - f_{\vec{w}, b}(\vec{x}^{(i)}))$$

We have:

$$\mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)}) = \begin{cases} -\log(f_{\vec{w}, b}(\vec{x}^{(i)})) & \text{if } y^{(i)} = 1 \\ -\log(1 - f_{\vec{w}, b}(\vec{x}^{(i)})) & \text{if } y^{(i)} = 0 \end{cases} \quad (4.12)$$

Therefore:

$$\mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)}) = -y^{(i)} \log(f_{\vec{w}, b}(\vec{x}^{(i)})) - (1 - y^{(i)}) \log(1 - f_{\vec{w}, b}(\vec{x}^{(i)}))$$

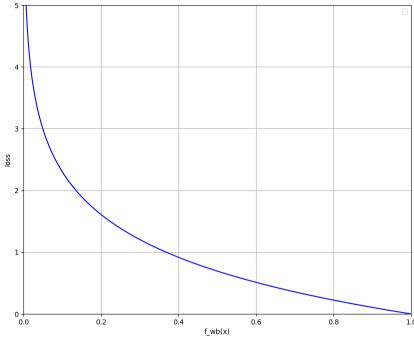


Figure 4.10:  $-\log(f_{\vec{w}, b}(\vec{x}^{(i)}))$

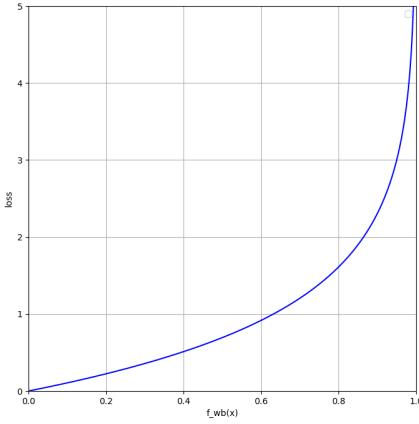


Figure 4.11:  $\log(1 - f_{\vec{w}, b}(\vec{x}^{(i)}))$

#### 4.5.4 Cost function

Cost  $\mathcal{J}(\vec{w}, b)$  is a measure of the losses over the training set

$$\mathcal{J}(\vec{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} [\mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)})]$$

Our objective is to minimise the cost function and this problem can be reasonably solved by applying gradient descent.

#### 4.5.5 Gradient descent

Gradient descent is a good way to help us reduce the cost of the model by picking the weight and bias parameters that give the smallest possible value of  $\mathcal{J}(\vec{w}, b)$ . In detail, we will start with some  $w$  and  $b$ , for instance, we can set  $\vec{w} = \vec{0}$  and  $b = 0$ . After that, we keep changing  $w$ ,  $b$  to reduce  $\mathcal{J}(\vec{w}, b)$  until we level off at the minimum of cost  $\mathcal{J}(\vec{w}, b)$ . Let get parameter

$w_j$  as an example, the partial derivatives  $\frac{\partial \mathcal{J}(w, b)}{\partial w_j}$  at a point on the cost function  $\mathcal{J}(\vec{w}, b)$  curve is illustrated as slope of the tangent line at that point.

**Case 1:** when the optimization point of  $w_j$  for minimum the cost  $\mathcal{J}(\vec{w}, b)$  is less than the current  $w_j$ , we want to move  $w_j$  to the left, so  $w_j$  should minus a positive number to end up with a new smaller value  $w_j$ .

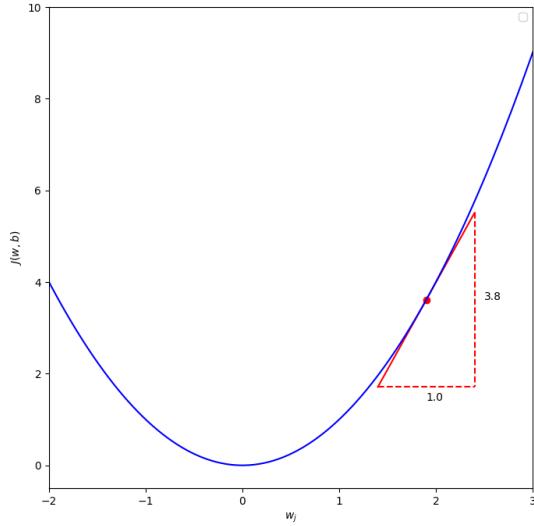


Figure 4.12: positive slope

We have learning rate  $\alpha$  is a positive number and the tangent line is pointing up to the right that lead the slope to be positive, which means that the derivative  $\frac{\partial \mathcal{J}(w, b)}{\partial w_j}$  is a positive number.

$$w_j = w_j - \alpha \frac{\partial \mathcal{J}(\vec{w}, b)}{\partial w_j}$$

$$w_j = w_j - (\text{positive number}) * (\text{positive number})$$

$$w_j = w_j - (\text{positive number})$$

then  $w_j$  will become smaller

thus,  $w_j$  move to the left

**Case 2:** when the optimization point of  $w_j$  for minimum the cost  $\mathcal{J}(\vec{w}, b)$  is greater than the current  $w_j$ , we want to move  $w_j$  to the right, so  $w_j$  should minus a negative number to end up with a new larger value  $w_j$ .

We have learning rate  $\alpha$  is a positive number and the tangent line is pointing down to the

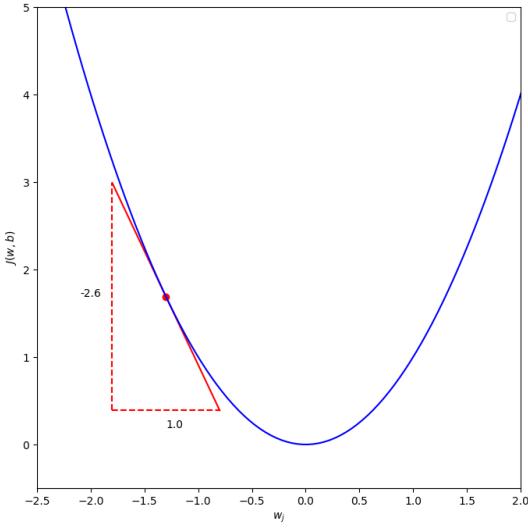


Figure 4.13: negative slope

right that lead the slope to be negative, which means that the derivative  $\frac{\partial \mathcal{J}(w, b)}{\partial w_j}$  is a negative number.

$$w_j = w_j - \alpha \frac{\partial \mathcal{J}(\vec{w}, b)}{\partial w_j}$$

$$w_j = w_j - (\text{positive number}) * (\text{negative number})$$

$$w_j = w_j - (\text{negative number})$$

then  $w_j$  will become larger

thus,  $w_j$  move to the right

In both cases, we take the same function  $w_j = w_j - \alpha \frac{\partial \mathcal{J}(w, b)}{\partial w_j}$ . Therefore, we will change simultaneously all parameters w and b by applying the same method that the parameters are updated to be minus the learning rate times the derivative term.

$$\begin{aligned}
\frac{\partial \mathcal{J}(\vec{w}, b)}{\partial w_j} &= \frac{1}{m} \sum_{i=0}^{m-1} (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)}) x_j^{(i)} \\
\frac{\partial \mathcal{J}(\vec{w}, b)}{\partial b} &= \frac{1}{m} \sum_{i=0}^{m-1} (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)}) \\
w_j &= w_j - \alpha \frac{\partial \mathcal{J}(\vec{w}, b)}{\partial w_j} && \text{for } j := 0..n-1 \\
b &= b - \alpha \frac{\partial \mathcal{J}(\vec{w}, b)}{\partial b}
\end{aligned}$$

Nevertheless, the gradient descent can make the cost  $\mathcal{J}(\vec{w}, b)$  stuck at a local minimum, which is not the global minimum so that we should differentiate the starting point of parameters  $\vec{w}$  and  $b$ . Let's get  $\mathcal{J}(w)$  as an example:

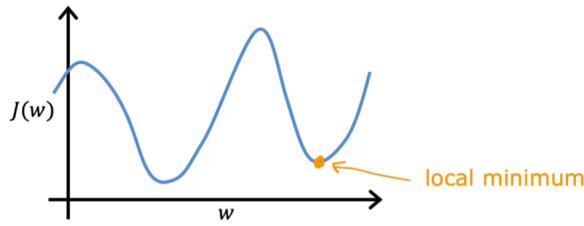


Figure 4.14: Stuck at a local minimum

#### 4.5.6 Learning

##### Learning rate

As choosing a learning rate affects intensively the efficiency when implementing gradient descent, we will go through this concept. The learning rate  $\alpha$  is a tuning positive number that settles the variation of the parameters  $w$  and  $b$  at each iteration while the cost function  $\mathcal{J}(\vec{w}, b)$  approaches the minimum. If  $\alpha$  is too small, the gradient descent may slowly move the cost toward the minimum and the program break before the cost reaches end up at the local minimum. However, a large  $\alpha$  can lead the cost to overshoot and never reach minimum. We ordinarily set  $\alpha = 10^{-3}$ , although we should manipulate the learning rate so as to get the appropriate value for the learning model. With an easy approach, we can set  $\alpha = 1$  as a high learning at the beginning, if the cost is divergent, we moderate  $\alpha$  by reducing it by 3.

## Learning process

---

### Algorithm 1 Logistic Regression

---

#### Iterative Process

- Given feature  $\vec{x}$  and target  $y$  with  $m$  training examples,  $n$  features
- Set the learning rate array  $A = [1, 3.10^{-1}, 10^{-1}, 3.10^{-2}, 10^{-2}, 3.10^{-3}, 10^{-3}, 3.10^{-4}]$
- For  $\alpha$  equals to each value in array A, starting from the highest  $\alpha = 1$ :

– Initialize weight  $\vec{w}$  and bias  $b$

– Iterate (typically 1000 times) gradient descent to minimize cost:

$$* \text{ Calculate } f_{\vec{w}, b}(\vec{x}) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x} + b)}}$$

$$* \text{ Compute loss } \mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)}) \text{ for } i := 0..m-1$$

$$\mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)}) = -y^{(i)} \log(f_{\vec{w}, b}(\vec{x}^{(i)})) - (1 - y^{(i)}) \log(1 - f_{\vec{w}, b}(\vec{x}^{(i)}))$$

\* Get the current cost as the average of the losses:

$$\mathcal{J}(\vec{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} [\mathcal{L}(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)})]$$

\* If the change in  $\mathcal{J}(\vec{w}, b)$   $\leq$  stopping threshold we stop the gradient descent

\* Else update  $\vec{w}$  and  $b$  to reduce the cost:

$$\frac{\partial \mathcal{J}(\vec{w}, b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

$$\frac{\partial \mathcal{J}(\vec{w}, b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)})$$

$$w_j = w_j - \alpha \frac{\partial \mathcal{J}(\vec{w}, b)}{\partial w_j} \quad \text{for } j := 0..n-1$$

$$b = b - \alpha \frac{\partial \mathcal{J}(\vec{w}, b)}{\partial b}$$

– If the cost  $\mathcal{J}(\vec{w}, b)$  is convergent, differentiate the starting point of the parameters

$\vec{w}$  and  $b$  to ensure this is the global minimum

– Else if the cost  $\mathcal{J}(\vec{w}, b)$  is divergent, continue moderate the learning rate

#### Output

$f_{\vec{w}, b}(\vec{x})$  comparing to the given threshold to get  $\hat{y}$  as the prediction

---

## 4.6 Decision Tree

### 4.6.1 Overview

The Decision Tree algorithm is a supervised learning method capable of handling both *classification* and *regression* tasks. It works by learning decision rules inferred from the training data to predict the target variable's value or class. Starting from the root, the algorithm sorts records down the tree based on attribute comparisons until reaching a terminal node that represents the prediction.

#### Types of Decision Trees

Decision trees are classified by the type of target variable:

- **Categorical Variable Decision Tree:** Used when the target variable is categorical.
- **Continuous Variable Decision Tree:** Used for continuous target variables.

#### Important Terminology

- **Root Node:** Represents the complete dataset, divided into sub-nodes.
- **Splitting:** Dividing a node into sub-nodes.
- **Decision Node:** A node that splits into further sub-nodes.
- **Leaf/Terminal Node:** A node with no further splits.
- **Pruning:** The process of removing sub-nodes to avoid overfitting.
- **Branch/Sub-Tree:** A sub-section of the entire tree.
- **Parent and Child Nodes:** The parent node divides into child nodes.

### 4.6.2 Working Mechanism of Decision Trees

Decision Trees work by making strategic splits based on attributes that maximize homogeneity in the sub-nodes. Multiple algorithms guide this splitting process, including:

- **ID3 (Iterative Dichotomiser 3):** Selects attributes with the highest information gain.
- **C4.5:** Successor of ID3, uses gain ratio to overcome information gain bias.
- **CART (Classification and Regression Tree):** Uses the Gini index for classification and regression tasks.

- **CHAID (Chi-square Automatic Interaction Detection)**: Uses Chi-square tests for classification trees.
- **MARS (Multivariate Adaptive Regression Splines)**: Useful for regression tasks.

### 4.6.3 Attribute Selection Measures

Selecting the root and internal nodes is based on *attribute selection measures* like:

- **Entropy and Information Gain**: Used to calculate the reduction in entropy before and after a split.
- **Gini Index**: Measures impurity for binary splits.
- **Gain Ratio**: Adjusts information gain for attributes with many distinct values.
- **Reduction in Variance**: Useful for continuous target variables.
- **Chi-Square**: Evaluates statistical significance in splits.

### 4.6.4 Information Gain

Information Gain (IG) is a metric used to measure the reduction in entropy after splitting a dataset on an attribute. It calculates how well a given attribute separates the training examples according to their target classification. Mathematically, it is defined as:

$$IG(T, X) = \text{Entropy}(T) - \sum_{j=1}^K \frac{|T_j|}{|T|} \text{Entropy}(T_j)$$

where  $T$  is the dataset before the split,  $T_j$  represents each subset generated by the split,  $K$  is the number of subsets, and  $\text{Entropy}(T)$  is the entropy of the dataset  $T$ . Information Gain is maximized when the attribute splits the data into the purest possible sub-groups, reducing overall entropy.

### 4.6.5 Gini Index

The Gini Index is another metric used to evaluate splits in the dataset, **particularly for binary splits**. It is defined as the *probability of incorrectly classifying a randomly chosen element from the dataset* if it were labeled according to the distribution of labels in the subset. The Gini Index for a node  $T$  is given by:

$$\text{Gini}(T) = 1 - \sum_{i=1}^C p_i^2$$

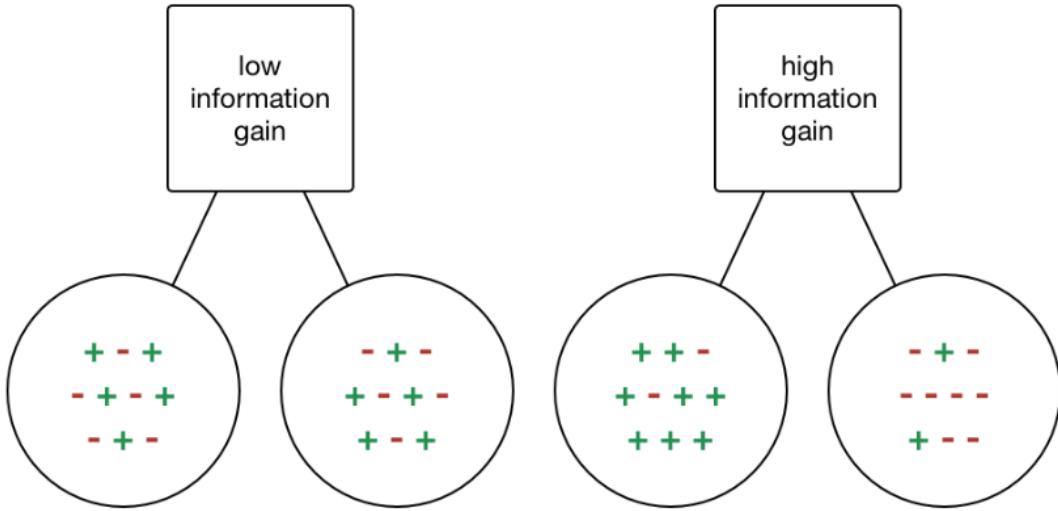


Figure 4.15: Information Gain

where  $C$  is the number of classes and  $p_i$  is the probability of selecting a sample with class  $i$ . A lower Gini Index indicates higher purity, meaning a more homogeneous subset.

#### 4.6.6 Comparison of Information Gain and Gini Index

Metric	Information Gain	Gini Index
Type of Split	Works best with multiple splits	Generally used for binary splits
Calculation	Based on entropy reduction	Based on probability of misclassification
Range	Non-negative, varies by dataset	Ranges from 0 to 0.5 for binary classification
Preference	Prefers attributes with many distinct values	Often simpler and computationally efficient
Bias	Biased towards attributes with more categories	Less biased towards highly categorical attributes

Table 4.3: Comparison between Information Gain and Gini Index in Decision Trees

#### 4.6.7 One-hot encoding of categorical features

If a feature has  $k > 2$  categorical values, change the feature to  $k$  binary features that have the value of 0 or 1.

#### 4.6.8 Continuous valued features

Choose the  $m - 1$  mid-points between the  $m$  examples as possible splits, and find the split that gives the highest information gain.

#### 4.6.9 Regression Trees

When choosing a split for a regression tree, we pick the feature with the highest reduction in variance.

$$\bar{x} = \frac{\sum x}{n}$$
$$V = \sigma^2 = \frac{\sum x_i^2 - n\bar{x}}{n-1}$$

$$\text{Reduction in Variance} = V(\text{root}) - (w^{\text{left}}V(\text{left}) + w^{\text{right}}V(\text{right}))$$

#### 4.6.10 Learning Process

---

##### Algorithm Decision Tree

---

###### Iterative Process

- All the labeled examples set in a node
- At each node:
  - Calculate information gain for all possible features and pick the one with the highest information gain
  - Split example according to selected feature, create left and right branches of the tree
  - Continue the recursive splitting process until stopping criteria is met

###### Stopping criteria

- The examples of a node are completely in the same class
  - The tree exceeds the maximum depth
  - Information gain from addition splits is less than threshold
  - Number of examples in a node is below threshold
-

## 4.7 Random Forest

### 4.7.1 Tree ensemble

We have the fact that decision tree algorithm highly sensitive to small changes in the data, changing just one training example causes the algorithm to come up with a different split at the root and then a totally different tree. Therefore, to make the algorithm less sensitive or more robust, we will build not just one decision tree, but a group of multiple decision trees, and we call that a tree ensemble.

Random forest is an algorithm that we apply tree ensemble, for each tree, the train set is sampled with replacement to create a new train set and features are randomly chosen.

### 4.7.2 Sampling with replacement train set

In the process of dataset generation for each tree, we apply sampling with replacement, which means after selecting randomly a train sample from the original train set, we put it back into the population then continue with another sample until we get the set of m train samples.

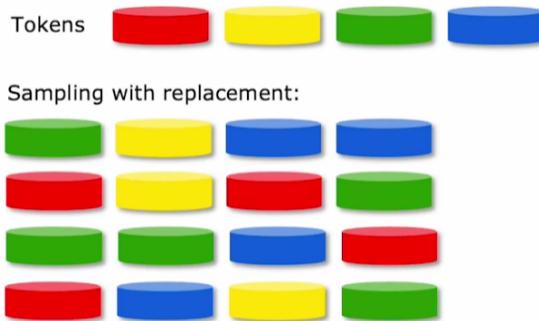


Figure 4.16: Sampling with replacement train set

### 4.7.3 Randomizing the feature choice

In case, we have totally  $n$  features, when choosing a feature to split at each node, we pick randomly a subset of  $k \leq n$ , typically  $k = \sqrt{n}$  features and let the algorithm only select a feature from the subset of these  $k$  features.

### 4.7.4 Learning Process

---

**Algorithm** Random Forest

---

**Iterative Process**

- Given  $m$  training examples and  $n$  features
  - Loop in limit times (typically ranging from 64 to 128), which is a number of generated trees:
    - Use sampling with replacement to generate a new training set of size  $m$
    - Train a decision tree on the new dataset with each node split by one of  $k = \sqrt{n}$  random features
    - The decision tree return as a vote
  - The final prediction is the highest vote for categorical target while it is the average of votes for continuous target.
- 

## 4.8 Noise and Outliers in Data Analysis

### 4.8.1 Definitions

**Noise** refers to any unwanted anomaly in the data that complicates the learning process and may make achieving zero error infeasible with a simple hypothesis class. Noise can arise in various forms:

- **Input Noise:** Imprecision in recording input attributes can cause shifts in data points within the input space.
- **Label Noise (Teacher Noise):** Errors in labeling may incorrectly categorize positive instances as negative, or vice versa.
- **Latent or Hidden Attributes:** Additional unobserved attributes may affect the label of an instance. These neglected attributes are modeled as a random component and considered part of the noise.

Noise introduces variability that can obscure true patterns in data, making the learning process more challenging.

**Outliers**, on the other hand, are data points that significantly deviate from other observations. They may arise from data entry errors, experimental inaccuracies, or rare events, but they can also contain meaningful information about the underlying system. Outliers are often referred to as *abnormalities, discordants, deviants, or anomalies*.

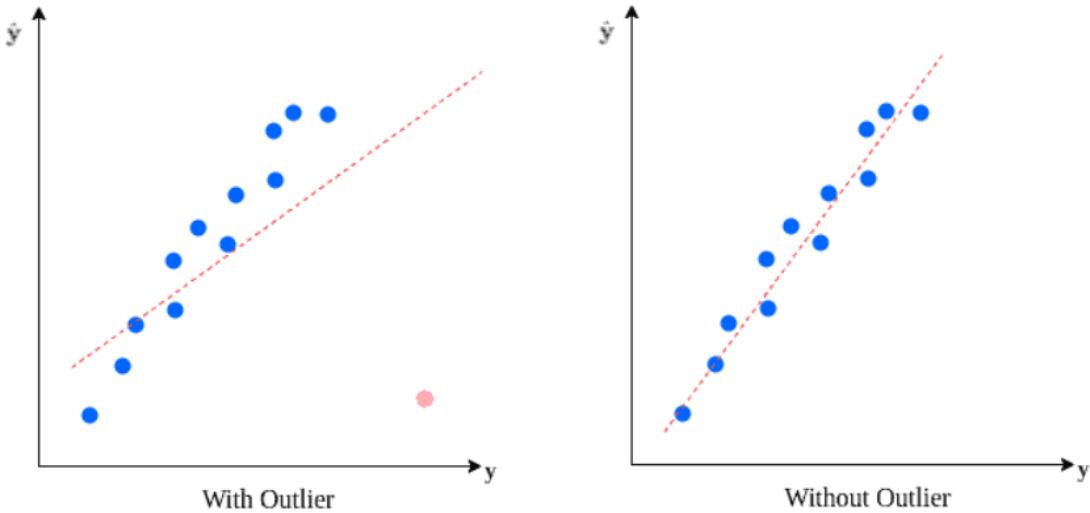


Figure 4.17: Outlier

Outliers are broader in scope compared to noise, as they include not only errors but also discordant data that may arise naturally from variations in the population or process. They are particularly valuable in fields such as fraud detection, intrusion detection, weather forecasting, and medical diagnosis, where identifying anomalies can lead to significant insights.

In the medical domain, common sources of outliers include equipment malfunctions, human errors, and patient-specific anomalies. For example, an abnormal blood test result could be due to pathology, medication intake, recent physical activity, or even improper sample handling. Evaluating whether an outlier represents an important finding or an error is crucial before any corrective action is taken.

### 4.8.2 Handling Noise and Outliers

#### Handling Noise

Noise can be managed through various data preprocessing techniques to improve model performance:

- **Data Cleaning:** Detecting and removing incorrect or inconsistent data, such as correcting input errors or discarding mislabeled instances.
- **Smoothing Techniques:** Techniques like moving averages or Gaussian filters can reduce the impact of noise in continuous data by smoothing variations.
- **Dimensionality Reduction:** Methods like Principal Component Analysis (PCA) can help reduce noise by focusing on the most informative components.
- **Robust Models:** Choosing models that are less sensitive to noise, such as robust regression or decision trees, which inherently ignore small fluctuations.

## Handling Outliers

Outliers require careful handling to avoid removing valuable information or introducing bias:

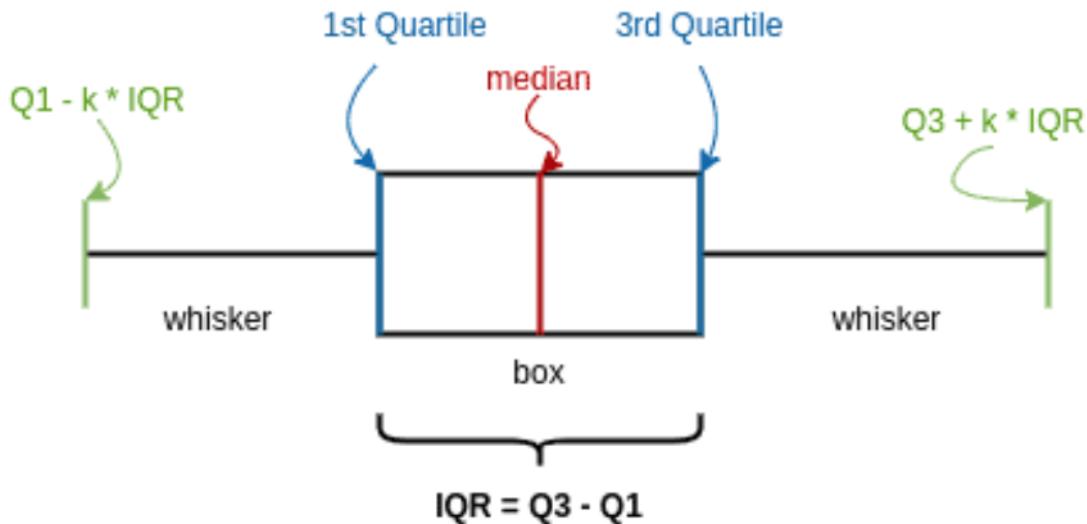


Figure 4.18: Inter Quartile Range

- **Statistical Methods:** Techniques such as Z-score, Interquartile Range (IQR), and modified Z-score help detect outliers based on statistical thresholds.
- **Clustering-Based Methods:** In complex datasets, clustering algorithms (e.g., DBSCAN) can identify outliers as points that do not belong to any cluster.
- **Model-Specific Detection:** Regression-based models are suitable for linearly correlated data, while neural networks or ensemble models may better handle nonlinear distributions.
- **Domain Knowledge:** Incorporating domain-specific insights, especially in fields like medicine, can help decide whether to retain, modify, or remove an outlier.

**Comment:** The choice of method depends on the data type, size, distribution, and the need for interpretability, especially in high-stakes fields like healthcare.

### 4.8.3 Comparison of Noise and Outliers

<b>Attribute</b>	<b>Noise</b>	<b>Outlier</b>
<b>Definition</b>	Random or irrelevant data that obscures true patterns.	Data points that significantly deviate from other observations, often called abnormalities or anomalies.
<b>Source</b>	Often due to measurement errors, environmental factors, or sensor inaccuracies.	Can result from data entry errors, rare events, or natural variations within the population.
<b>Impact on Model</b>	Generally reduces model accuracy by introducing random variability.	May affect model accuracy but can also represent meaningful insights or anomalies.
<b>Detection Methods</b>	Identified through signal processing, filtering, or data cleaning techniques.	Detected through statistical methods like Z-score, IQR, clustering, or regression analysis.
<b>Treatment</b>	Typically removed or smoothed to enhance model clarity.	Can be removed, retained, or corrected depending on their cause and potential significance.

Table 4.4: Comparison of Noise and Outliers in Data Analysis

## 4.9 Evaluation metrics

### 4.9.1 Confusion Matrix and Fundamental Metrics

Let's assume a system aims to detect whether essays are generated by an AI model (e.g., a Large Language Model, LLM) or human-written. A sample of 100 essays is analyzed, and the results are summarized as follows:

- 40 essays are generated by an LLM (positive cases).
- 60 essays are human-written (negative cases).
- The model predicts 45 essays as LLM-generated (positive prediction) and 55 as human-written (negative prediction).

Actual / Predicted	Predicted Positive (LLM-generated)	Predicted Negative (Human-written)	Total
Actual Positive (LLM-generated)	True Positive (TP) = 35	False Negative (FN) = 5	40
Actual Negative (Human-written)	False Positive (FP) = 10	True Negative (TN) = 50	60
Total	45	55	100

Table 4.5: Confusion matrix for the AI-generated text detection system.

The confusion matrix consists of four fundamental metrics that are used to evaluate the classifier:

- **TP (True Positive)**: The number of correctly classified LLM-generated essays.
- **TN (True Negative)**: The number of correctly classified human-written essays.
- **FP (False Positive)**: The number of human-written essays misclassified as LLM-generated. FP is also known as Type I error.
- **FN (False Negative)**: The number of LLM-generated essays misclassified as human-written. FN is also known as Type II error.

#### True Positive Rate (TPR) / Recall / Sensitivity

Recall shows the proportion of true positive predictions among all actual positive instances (LLM-generated essays) in the dataset. Here, TPR measures the proportion of actual LLM-generated essays correctly identified.

$$TPR = \text{Recall} = \frac{TP}{TP + FN} = \frac{35}{35 + 5} = \frac{35}{40} = 0.875 \quad (4.13)$$

#### False Positive Rate (FPR)

FPR shows the proportion of false positives among all actual negative instances (human-written essays). In this case, FPR represents the proportion of human-written essays incorrectly classified as LLM-generated.

$$FPR = \frac{FP}{FP + TN} = \frac{10}{10 + 50} = \frac{10}{60} = 0.167 \quad (4.14)$$

### Precision (Positive Predictive Value)

Precision measures the ratio of correctly classified LLM-generated essays (TP) to the total number of essays predicted to be LLM-generated (TP+FP).

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{35}{35 + 10} = \frac{35}{45} \approx 0.778 \quad (4.15)$$

### Accuracy

Accuracy represents the overall proportion of correct predictions.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{35 + 50}{100} = \frac{85}{100} = 0.85 \quad (4.16)$$

### F1 Score (F Measure)

The F1 score is the harmonic mean of precision and recall, balancing both metrics.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \approx 2 \times \frac{0.778 \times 0.875}{0.778 + 0.875} \approx 0.823 \quad (4.17)$$

### F-beta Score

$$F_\beta = (1 + \beta^2) \times \frac{\text{Precision} \times \text{Recall}}{\beta^2 \times \text{Precision} + \text{Recall}}$$

Evolve from F1 score, F-beta score is also called "weighted harmonic mean of precision and recall", where the parameter  $\beta$  controls the balance between precision and recall.

- If FP and FN both are important,  $\beta = 1$  (F1 score)
- If FP is more important than FN,  $\beta < 1$
- If FN is more important than FP,  $\beta > 1$

### Classification threshold

It is clear that a probabilistic classification model produces a probability score between 0 (totally human-written) and 1 (totally AI-written) for each essay, indicating the likelihood that it is AI-written. For instance, the model may assign a probability of 0.9 that an essay is AI-written. To determine the actual class label (human or AI), a classification threshold is applied.

- If the threshold is set to 0.5, any essay with a predicted probability above 0.5 is categorized as AI-written (class 1).
- Alternatively, a more conservative threshold of 0.8 might be chosen, only classifying essays with a probability exceeding 0.8 as AI-written.

The choice of classification threshold depends on the specific problem and the costs associated with different error types. For example, in detecting AI-written essays, a false positive (classifying a human-written essay as AI-written) may be less detrimental than a false negative (failing to detect an AI-written essay). In such cases, a lower threshold might be preferred to ensure higher recall, even if it leads to some false positives.

As you change the threshold, you will usually get new combinations of errors of different types (and new confusion matrices).

The selection of the classification threshold influences the model's error rate. A higher threshold increases the model's conservatism, assigning the label of AI-written only with greater confidence, but resulting in lower recall by detecting fewer true positive (AI-written) cases. Conversely, a lower threshold makes the model less strict, leading to higher recall, but potentially lower precision due to an increase in false positives (human-written essays classified as AI-written).

Modifying the classification threshold affects both the True Positive Rate (TPR) and False Positive Rate (FPR), causing them to change in the same direction. A higher TPR (recall) typically corresponds to a higher FPR, and vice versa.

Consider the following scenario where the recall (TPR) decreases as the decision threshold is set higher:

- Threshold of 0.5:  $\frac{800}{800 + 100} = 0.89$  (Recall for AI-written essays)
- Threshold of 0.8:  $\frac{600}{600 + 300} = 0.67$
- Threshold of 0.95:  $\frac{200}{200 + 700} = 0.22$

Similarly, the FPR also decreases:

- Threshold of 0.5:  $\frac{500}{500 + 8600} = 0.06$  (FPR, indicating the rate at which human-written essays are misclassified as AI-written)
- Threshold of 0.8:  $\frac{100}{100 + 9000} = 0.01$
- Threshold of 0.95:  $\frac{10}{10 + 9090} = 0.001$

## 4.9.2 Receiver Operating Characteristic (ROC) curve

The ROC curve was first developed by electrical engineers and radar engineers during World War II for detecting enemy objects in battlefields, starting in 1941, which led to its name ("receiver operating characteristic"). The ROC curve plots the True Positive Rate (TPR) against the False Positive rate (FPR) at various classification thresholds. Revising the previous section, we can derive TPR and FPR from a confusion matrix. The ROC curve is the most popular form of trade-off between TPR and FPR. Unless the model is near-perfect, we have to balance the two. As we try to increase the TPR (i.e., correctly identify more positive cases), the FPR may also increase (i.e., you get more false alarms). The more AI-written essays we want to detect, the more human-written essays which are falsely identified.

The ROC curve is a visual representation of this choice. Each point on the curve corresponds to a combination of TPR and FPR values at a specific decision threshold.

We will plot a ROC curve using scikit-learn's visualization API, RocCurveDisplay.

Listing 4.3: SVM vs Random Forest with ROC Curve

```
import matplotlib.pyplot as plt

from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import RocCurveDisplay
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

X, y = load_wine(return_X_y=True)
y = y == 2

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
svc = SVC(random_state=42)
svc.fit(X_train, y_train)

rfc = RandomForestClassifier(n_estimators=10, random_state=42)
rfc.fit(X_train, y_train)

ax = plt.gca()          # Get the current axis. Prevent the two ROC curves
                       # having different axes
svc_disp = RocCurveDisplay.from_estimator(svc, X_test, y_test, ax=ax,
                                           alpha=0.8)
rfc_disp = RocCurveDisplay.from_estimator(rfc, X_test, y_test, ax=ax,
                                           alpha=0.8)

plt.show()
```

The left side of the curve corresponds to the more "confident" thresholds: a higher threshold leads to lower recall and fewer false positive errors. The extreme point is when both recall and FPR are 0. In this case, there are no correct detections but also no false ones.

The right side of the curve represents the "less strict" scenarios when the threshold is low.

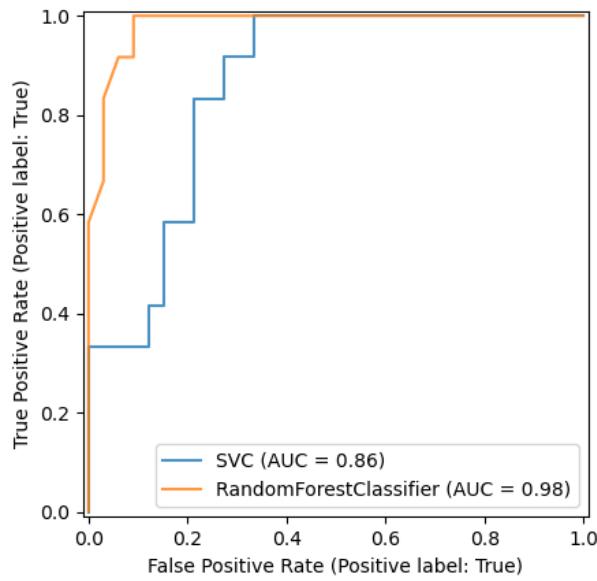


Figure 4.19: ROC Curve of Support Vector Machine vs Random Forest

Both recall and False Positive rates are higher, ultimately reaching 100%. If you put the threshold at 0, the model will always predict a positive class: both recall, and the FPR will be 1.

When you increase the threshold, you move left on the curve. If you decrease the threshold, you move to the right.

The ROC area under the curve (ROC AUC) is basically the ratio between the area under the curve and the rectangle formed by the TPR and FPR axes.

The perfect scenario would be the model identifying all cases correctly and never giving false alarms.



Figure 4.20: A Perfect ROC "Curve"

The worst scenario would be our model giving literally random results, leading to the ROC

curve begin a diagonal line connecting (0,0) to (1,1). For a random classifier, the TPR is equal to the FPR because it makes the same number of true and false positive predictions for any threshold value. As the classification threshold changes, the TPR goes up or down in the same proportion as the FPR.

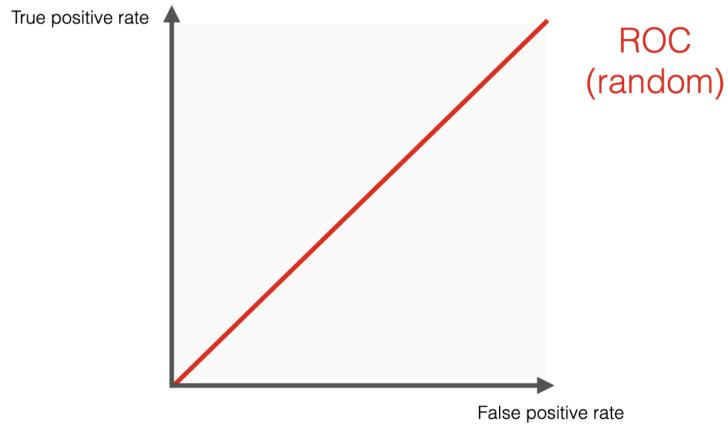


Figure 4.21: A totally random ROC "Curve"

Listing 4.4: Randomize the input and output to simulate a random classifier

```

1 import numpy as np
2
3 # Generate an array of 100 points
4 y_true = np.random.randint(0, 2, size=100) # 100 true labels (0 or 1)
5 y_scores = np.random.rand(100) # 100 random scores between 0 and 1
6
7 import matplotlib.pyplot as plt
8
9 from sklearn.datasets import load_wine
10 from sklearn.ensemble import RandomForestClassifier
11 from sklearn.metrics import RocCurveDisplay, auc, roc_curve
12 from sklearn.model_selection import train_test_split
13 from sklearn.svm import SVC
14
15 fpr, tpr, _ = roc_curve(y_true, y_scores)
16 roc_auc = auc(fpr, tpr)
17
18 # Create ROC Curve plot
19 roc_display = RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc,
20     estimator_name='Example Model')
21
22 # Plot and save
23 roc_display.plot()
24 plt.savefig('ROC AUC random.png') # Save the ROC curve plot as an image

```

Most real-world models will fall somewhere between the two extremes. The better the model can distinguish between positive and negative classes, the closer the curve is to the top left

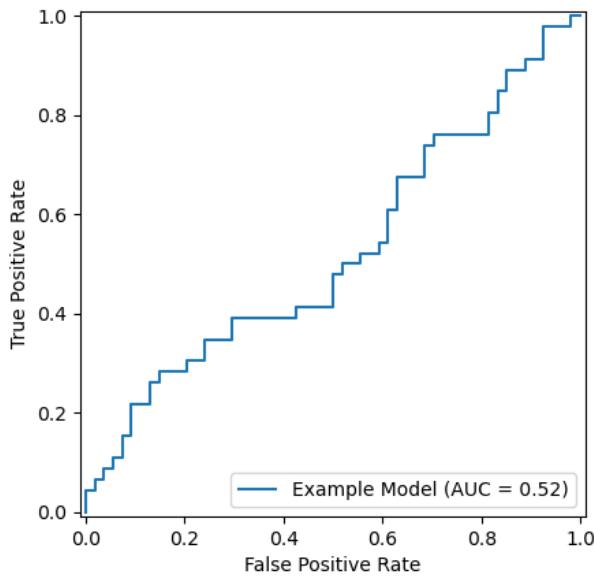


Figure 4.22: Simulated ROC curve result of a random classifier

corner of the graph. As a rule of thumb, a ROC AUC score above 0.8 is considered good, while a score above 0.9 is considered great.

### Evaluation

The intuition behind ROC AUC is that it measures how well a binary classifier can distinguish or separate between the positive and negative classes. It reflects the probability that the model will correctly rank a randomly chosen positive instance higher than a random negative one.

ROC AUC is a powerful metric for evaluating binary classification models. However, its usefulness depends on the context and the specific goals of your model.

*When to Use ROC AUC:*

- **Comparing Multiple Models:** ROC AUC is valuable during model training to compare the overall performance of multiple models. Since it summarizes the performance across different thresholds in a single number, it is easy to use when you want to identify the best-performing model in general terms.
- **Ranking Predictions by Confidence:** If your primary goal is to rank predictions by how confident the model is, rather than making hard predictions at a fixed threshold, ROC AUC is ideal. It measures how well the model separates the positive class from the negative class, regardless of the decision threshold.
- **Model Monitoring in Production:** During production model monitoring, ROC AUC can track whether the model still separates positive and negative classes effectively, especially when class distributions change over time. This can provide an early

warning of deteriorating model performance before more granular metrics like precision and recall are impacted.

*When Not to Use ROC AUC:*

- **When Costs of Errors Vary:** ROC AUC does not take into account the costs of false positives and false negatives, which may be very different in certain applications (e.g., medical diagnoses or fraud detection). If minimizing one type of error (false positives or false negatives) is more critical, metrics like precision, recall, or F1-score are better suited.
- **Severe Class Imbalance:** When the positive class is very rare, ROC AUC can give a false sense of good model performance, as it may reflect that the model is correctly predicting negatives (which dominate the data), but it may still be missing most positives. In such cases, the precision-recall curve and metrics like precision, recall, or F1-score are more appropriate.
- **Optimizing for a Specific Threshold:** If you need to set a specific threshold to balance the trade-offs between false positives and false negatives (e.g., in cases with asymmetric costs of errors), ROC AUC will not help you. Instead, you'll need to analyze precision, recall, and possibly tune the threshold manually to minimize the cost of errors.

Metric	When to Use	Strengths	Weaknesses
<b>Precision</b>	When false positives (human-written essays misclassified as AI-written) are costly. You want to ensure that positive predictions (AI-written) are correct.	Focuses on minimizing false positives; high precision means fewer human-written essays are wrongly flagged as AI-written.	Ignores false negatives, meaning it may miss some AI-written essays (if recall is low).
<b>Recall</b>	When false negatives (AI-written essays misclassified as human-written) are costly. You want to capture all AI-written essays.	Focuses on minimizing false negatives; high recall means most AI-written essays are detected.	Ignores false positives, so it may lead to more human-written essays being misclassified as AI-written.
<b>Accuracy</b>	When the dataset is balanced (equal numbers of AI and human-written essays) and both types of errors (false positives and false negatives) are equally important.	Provides an overall measure of correctness. Easy to understand and interpret.	Can be misleading in imbalanced datasets (e.g., if there are far fewer AI-written essays). May not be useful when one type of error is more important.
<b>F1-Score</b>	When you need a balance between precision and recall, especially in the case of imbalanced datasets (e.g., fewer AI-written essays).	Balances both false positives and false negatives; useful when both precision and recall are important.	Less intuitive than precision or recall individually. Does not consider different costs of errors.
<b>ROC AUC</b>	When you want to evaluate the overall performance of the model across different classification thresholds, particularly for ranking tasks.	Summarizes performance across all thresholds; useful for comparing multiple models.	Does not account for the costs of false positives and false negatives. Can be misleading with severe class imbalance.

Table 4.6: Evaluation Metric Comparison

Considering the imbalanced nature of our dataset (too many human-written essays compared to AI-written), and that they have equal importance to the model, the F1-score is the best choice amongst the given metrics.

More info about why ROC AUC is insufficient for imbalanced dataset:

<https://www.kaggle.com/code/lct14558/imbalanced-data-why-you-should-not-use-roc-curve>

### 4.9.3 Type I Error vs Type II Error: An Inevitable Tradeoff

Type I and Type II errors play critical roles in various fields, including law and medicine, where the consequences of each error can significantly impact lives and society. Let's explore the tradeoffs and implications in these domains.

#### Case study 1: Law

- **Type I Error (False Positive):** Convicting an innocent person, also known as a "false conviction."
- **Type II Error (False Negative):** Failing to convict a guilty person, often referred to as a "false acquittal."

*Implications of Type I Errors:* A Type I error in law undermines trust in the justice system, as it leads to wrongful punishment. An innocent person suffers loss of freedom, reputation, and often mental well-being, and society bears the cost of their wrongful incarceration.

*Implications of Type II Errors:* A Type II error lets a guilty person go free, potentially endangering society by allowing harmful individuals to re-offend. It can also diminish the perceived deterrent effect of the justice system.

Most legal systems operate under the principle that "it is better that ten guilty persons escape than one innocent suffer." This reflects a strong preference to avoid Type I errors, prioritizing the protection of innocent individuals over convicting the guilty.

Lie detectors are sometimes used in criminal investigations, and both errors can have significant consequences:

- **Type I Error (False Positive):** An honest person is falsely labeled as lying, leading to suspicion or investigation.
- **Type II Error (False Negative):** A deceptive individual is classified as truthful, potentially hindering justice.

*Implications of Type I Error:* False positives in lie detection can lead to unnecessary legal pressure on innocent individuals, potentially affecting mental health and biasing judicial outcomes.

*Implications of Type II Error:* False negatives allow deceptive individuals to avoid detection, potentially hindering the pursuit of justice.

Often, there is a preference to avoid Type I errors to prevent wrongful suspicion of innocent individuals.

#### Case study 2: Medicine

In medicine, Type I and Type II errors relate to the diagnosis and treatment of diseases, where both types of errors can have serious consequences:

- **Type I Error (False Positive):** Diagnosing a patient with a condition they do not have, leading to unnecessary treatment.
- **Type II Error (False Negative):** Failing to diagnose a condition that is actually present, resulting in a lack of necessary treatment.

*Implications of Type I Errors:* A Type I error may lead to unnecessary treatments, side effects, psychological stress, and medical costs. For instance, diagnosing a patient with cancer who doesn't have it can lead to anxiety and harmful treatments.

*Implications of Type II Errors:* A Type II error may delay treatment, worsening the patient's condition and potentially reducing recovery chances. For example, failing to diagnose cancer early can make the disease harder to treat or even fatal.

In cases of life-threatening conditions (e.g., cancer, heart disease), doctors often prioritize avoiding Type II errors to ensure early detection, tolerating more false positives (Type I errors) as a result.

**Case study 3: AI in Academic Research Preference Against Type I Error:** There may be a preference to avoid Type I errors to protect researchers from undue scrutiny, given the reputational risks.

**Implications of Type I Error:** False positives may discourage researchers from using advanced tools or even language assistance that could be mistakenly flagged, limiting their productivity or creativity.

**Implications of Type II Error:** False negatives, however, can allow academic misconduct to go undetected. When AI-generated content is mistakenly classified as human-written, it can lead to an erosion of academic standards, as well as skewed or unreliable research findings, particularly if such content is submitted as original work or influences peer-reviewed research.

**Preference Against Type I Error:** In many academic contexts, there may be a preference to avoid Type I errors to protect innocent researchers from undue scrutiny or accusations. Labeling a human-authored work as AI-generated can lead to distrust in the detection system and harm honest researchers' reputations. However, excessive Type II errors could undermine research integrity if AI-generated work is mistaken for human-authored content.

#### 4.9.4 Impossibility Results for Reliable Detection of AI-Generated Text

In **probability theory**, the **total variation distance** is a distance measure for probability distributions. It is an example of a **statistical distance** metric and is sometimes called the *statistical distance*, *statistical difference* or *variational distance*.

##### Definition

Consider a measurable space  $(\Omega, \mathcal{F})$  and probability measures  $P$  and  $Q$  defined on  $(\Omega, \mathcal{F})$ . The total variation distance between  $P$  and  $Q$  is defined as

$$\delta(P, Q) = \sup_{A \in \mathcal{F}} |P(A) - Q(A)|.$$

This is the largest absolute difference between the probabilities that the two probability distributions assign to the same event.

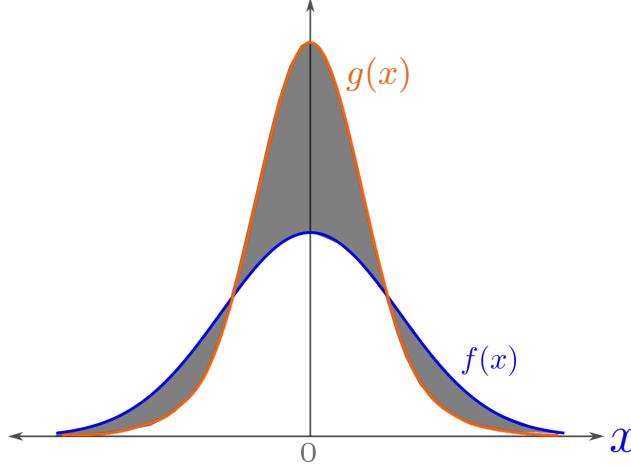


Figure 4.23: Total variation distance is half the absolute area between the two curves: Half the shaded area above.

In the following theorem, we formalize the above statement by showing an upper bound on the area under the ROC curve of an arbitrary detector in terms of the total variation distance between the distributions for AI and human-generated text. This bound indicates that as the distance between these distributions diminishes, the ROC AUC bound approaches  $1/2$ , which represents the baseline performance corresponding to a detector that randomly labels text as AI or human-generated. We define  $\mathcal{M}$  and  $\mathcal{H}$  as the text distributions produced by an AI model and humans, respectively, over the set of all possible text sequences  $\Omega$ . We use  $TV(\mathcal{M}, \mathcal{H})$  to denote the total variation distance between these two distributions and a function  $D : \Omega \rightarrow \mathbb{R}$  that maps every sequence in  $\Omega$  to a real number. Sequences are classified into AI and human-generated by applying a threshold  $\gamma$  on this number. By adjusting the parameter  $\gamma$ , we can tune the sensitivity of the detector to AI and human-generated texts to obtain an ROC curve.

**Theorem 1.** *The area under the ROC of any detector  $D$  is bounded as*

$$\text{ROC AUC}(D) \leq \frac{1}{2} + TV(\mathcal{M}, \mathcal{H}) - \frac{TV(\mathcal{M}, \mathcal{H})^2}{2}.$$

**Proof.** The ROC is a plot between the true positive rate (TPR) and the false positive rate (FPR) which are defined as follows:

$$\text{TPR}_\gamma = \mathbb{P}_{s \sim \mathcal{M}}[D(s) \geq \gamma]$$

and

$$\text{FPR}_\gamma = \mathbb{P}_{s \sim \mathcal{H}}[D(s) \geq \gamma],$$

where  $\gamma$  is some classifier parameter. We can bound the difference between the  $\text{TPR}_\gamma$  and the  $\text{FPR}_\gamma$  by the total variation between  $\mathcal{M}$  and  $\mathcal{H}$ :

$$|\text{TPR}_\gamma - \text{FPR}_\gamma| = |\mathbb{P}_{s \sim \mathcal{M}}[D(s) \geq \gamma] - \mathbb{P}_{s \sim \mathcal{H}}[D(s) \geq \gamma]| \leq TV(\mathcal{M}, \mathcal{H}) \quad (1)$$

$$\text{TPR}_\gamma \leq \text{FPR}_\gamma + TV(\mathcal{M}, \mathcal{H}). \quad (2)$$

Since the  $\text{TPR}_\gamma$  is also bounded by 1 we have:

$$\text{TPR}_\gamma \leq \min(\text{FPR}_\gamma + TV(\mathcal{M}, \mathcal{H}), 1). \quad (3)$$

Denoting  $\text{FPR}_\gamma$ ,  $\text{TPR}_\gamma$ , and  $TV(\mathcal{M}, \mathcal{H})$  with  $x$ ,  $y$ , and  $tv$  for brevity, we bound the ROC AUC as follows:

$$\begin{aligned} \text{ROC AUC}(D) &= \int_0^1 y \, dx \leq \int_0^1 \min(x + tv, 1) \, dx \\ &= \int_0^{1-tv} (x + tv) \, dx + \int_{1-tv}^1 dx \\ &= \left[ \frac{x^2}{2} + tvx \right]_0^{1-tv} + [x]_{1-tv}^1 \\ &= \frac{(1-tv)^2}{2} + tv(1-tv) + tv \\ &= \frac{1}{2} + \frac{tv^2}{2} + tv - tv + \frac{tv^2}{2} \\ &= \frac{1}{2} + tv - \frac{tv^2}{2}. \end{aligned}$$

□

For a detector to have a good performance (say,  $ROCAUC \geq 0.9$ ), the distributions of human and AI-generated texts must be very different from each other (total variation  $> 0.5$ ). As the two distributions become similar (say,  $totalvariation \leq 0.2$ ), the performance of even the bestpossible detector is not good ( $ROCAUC < 0.7$ ). This shows that distinguishing the text produced by a language model from a human-generated one is a fundamentally difficult task.

In a recent paper, Feizi analyzed two types of errors that affect the reliability of AI text detectors: type I errors, where human-written text is mistakenly identified as AI-generated, and type II errors, where AI-generated text goes undetected.

Feizi, who holds a joint appointment at the University of Maryland Institute for Advanced Computer Studies, pointed out that commonly available tools like paraphrasers can contribute to type II errors by altering AI-generated text in a way that allows it to evade detection. A notable example of a type I error recently gained attention online when AI detection software incorrectly flagged the U.S. Constitution as AI-generated, underscoring the potential for such technology to make glaring mistakes.

These errors can have serious consequences. Mistakes made by AI detectors can be especially damaging when used by authorities like educators or publishers, who may accuse students and other creators of using AI when, in fact, their work is original. Such accusations are often difficult to dispute, and when proven false, they can damage the reputations of the companies and developers behind the faulty AI detectors. Additionally, even large language models (LLMs) protected by watermarking schemes remain vulnerable to spoofing attacks, where malicious actors can infer hidden watermarks and add them to human-created text, causing it to be incorrectly flagged as AI-generated. These risks, Feizi argues, highlight the need for caution in relying solely on AI detectors to verify the authenticity of human-created content.

Feizi further emphasized the fundamental limitations of these detectors. He explained that, theoretically, it is impossible to reliably determine whether a random sentence was written by a human or generated by AI, given the close similarity in distribution between human and AI-generated content. This challenge is compounded by the sophistication of modern LLMs and the tools available to circumvent detection, like paraphrasers or spoofing techniques.

Our experiments show that paraphrasing the LLM outputs helps evade these detectors effectively. Moreover, our theory demonstrates that for a sufficiently advanced language model, even the best detector can only perform marginally better than a random classifier. This means that for a detector to have both low type-I and type-II errors, it will have to trade off the LLM's performance.

The line between human and AI-generated content is becoming increasingly blurred due to these variables, Feizi noted, and there is an inherent upper limit to the effectiveness of AI detectors. Consequently, he argues, it is unlikely that highly reliable detectors for AI-generated content will be developed in the foreseeable future.

## 4.10 Sampling Methods in Machine Learning

Sampling is a critical process in statistical analysis and machine learning, where a subset of a population or dataset is selected for analysis. Sampling can be broadly categorized into **probability sampling** and **non-probability sampling**. Sampling can also occur **with or without replacement**, an important distinction that defines whether a selected element can be chosen more than once in a single sample.

### 4.10.1 Probability Sampling

In probability sampling, each member of the population has a known, non-zero probability of being selected. This approach is often preferred for creating representative samples.

#### Simple random sampling

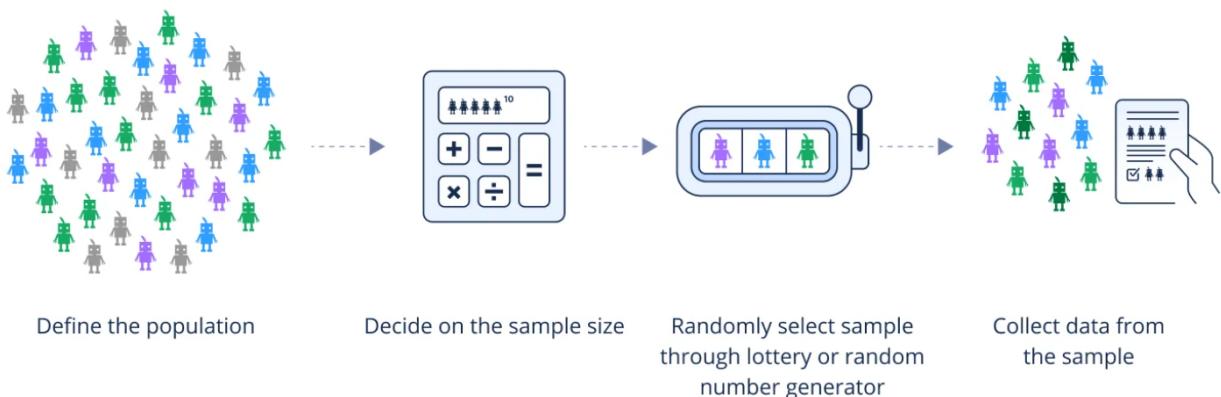


Figure 4.24: Simple Random Sampling

- **Simple Random Sampling:** Each member of the population has an equal chance of selection. Can be conducted *with or without replacement*.

## Systematic sampling

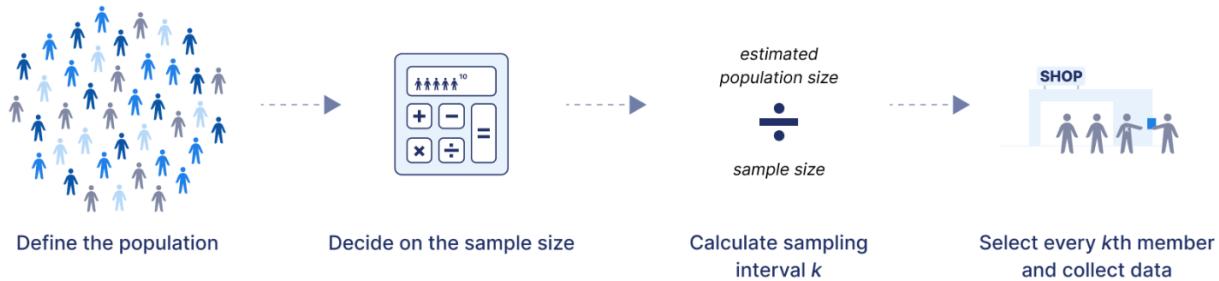


Figure 4.25: Systematic Sampling

- **Systematic Sampling:** Every  $k$ -th member is chosen from a list after a random starting point. This method is simple to implement but may introduce bias if the list has patterns.

## Stratified sampling

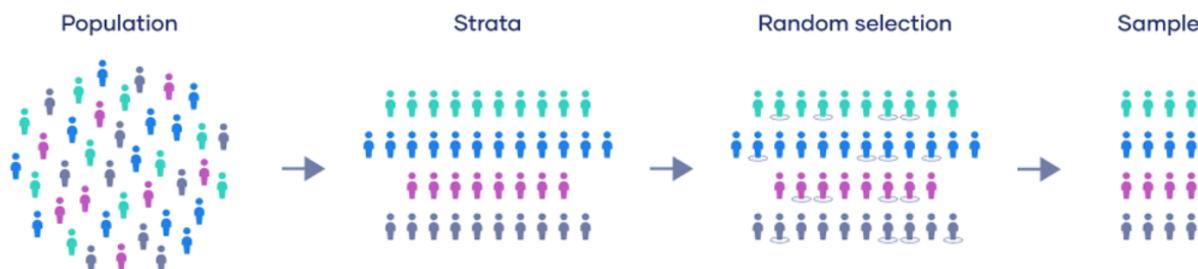


Figure 4.26: Stratified Sampling

- **Stratified Sampling:** The population is divided into strata (e.g., age groups), and random samples are drawn from each stratum. Ensures representation across key subgroups but requires knowledge of population characteristics.

## Cluster sampling

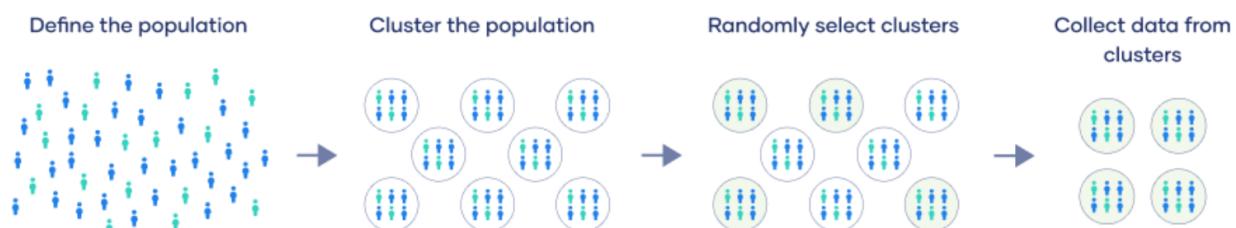


Figure 4.27: Cluster Sampling

- **Cluster Sampling:** The population is divided into clusters (e.g., geographic regions), and entire clusters are randomly selected. Useful for large, dispersed populations, but clusters may not be homogeneous, impacting representativeness.
- **Multistage Sampling:** A combination of the above techniques, often involving cluster sampling at an initial stage, followed by random or stratified sampling within clusters. Complex but effective for large-scale studies.

#### 4.10.2 Non-Probability Sampling

In non-probability sampling, some members of the population have zero chance of selection, often leading to bias.

- **Convenience Sampling:** Selection based on ease of access. Inexpensive but may not represent the entire population accurately.
- **Judgmental (or Purposive) Sampling:** The researcher selects participants based on their judgment of who would be most informative. Useful for expert opinion but lacks generalizability.
- **Quota Sampling:** The sample is designed to reflect specific characteristics of the population but without random selection within groups, which may introduce bias.
- **Snowball Sampling:** Participants recruit other participants, useful for hard-to-reach populations but may lead to bias if initial contacts are not representative.

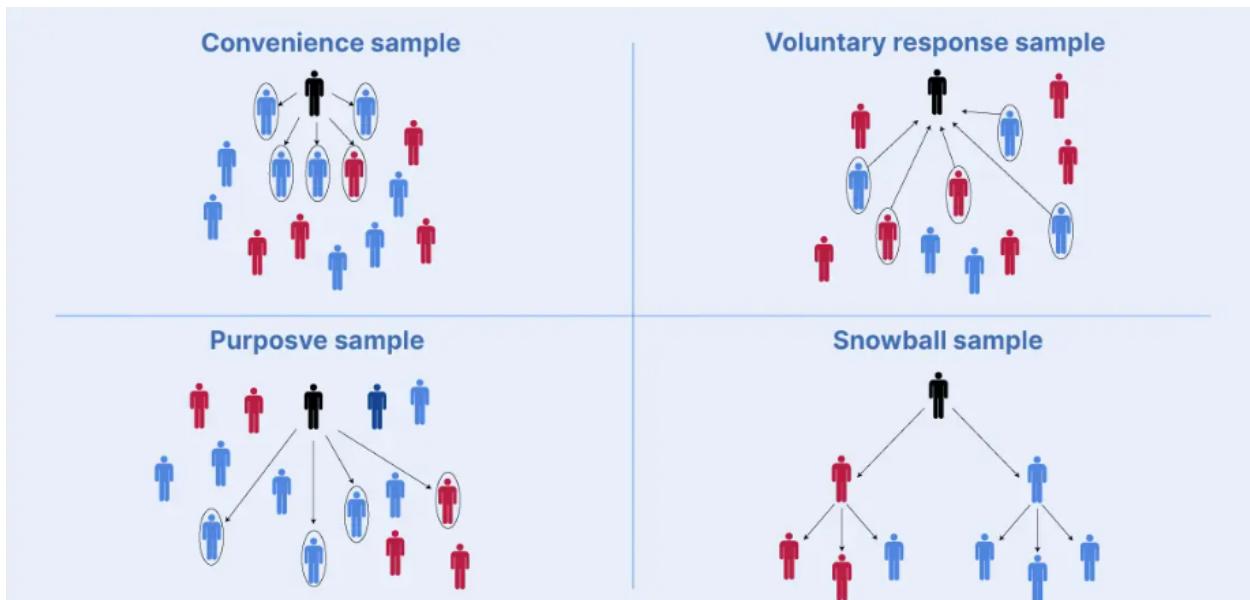


Figure 4.28: Non-Probability Sampling

#### 4.10.3 Sampling with and without Replacement

Sampling methods can be performed either **with replacement** or **without replacement**:

- **With Replacement:** Each selected element is returned to the population, allowing it to be chosen multiple times. This approach is useful in resampling techniques, such as bootstrapping.
- **Without Replacement:** Each selected element is removed from the population, preventing it from being selected again in the same sample. This approach ensures a diverse sample without repeated elements.

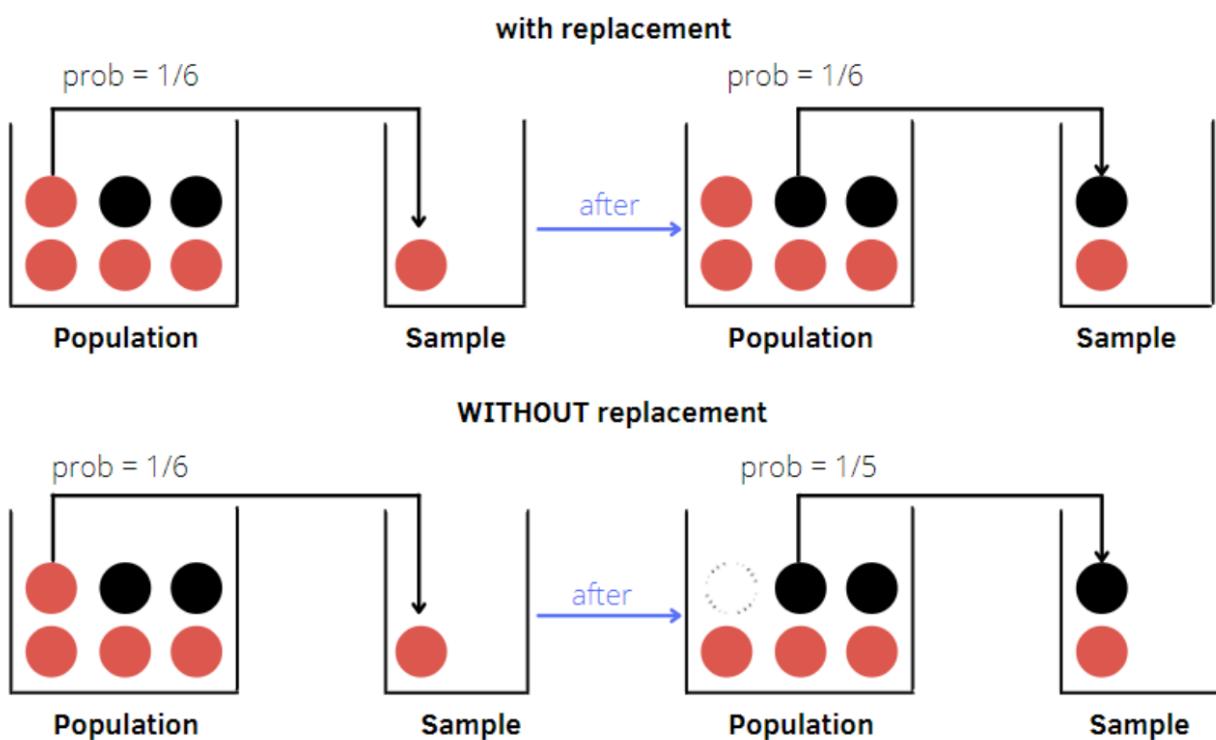


Figure 4.29: Sampling with and without Replacement

#### 4.10.4 Feature Randomization

Feature randomization is a technique often used in ensemble learning methods, particularly in decision tree-based models like Random Forests. In feature randomization, only a subset of features is selected randomly at each split in the decision tree construction. This process has several benefits:

- **Reduces Overfitting:** Randomizing features at each split reduces the likelihood that the model becomes overly dependent on any single feature, enhancing generalizability.

- **Increases Model Diversity:** By using different subsets of features for each tree, Random Forests can generate diverse decision trees, which helps improve ensemble robustness.
- **Improves Computational Efficiency:** Limiting the number of features considered at each split speeds up the model training process, particularly with high-dimensional datasets.

In feature randomization, the commonly used heuristic is to select  $k = \sqrt{n}$  features, where  $n$  is the total number of features. This choice strikes a balance between model complexity and the need for diversity, allowing each tree to focus on different aspects of the data without becoming too complex.

#### 4.10.5 Feature Flagging

Feature flagging is a technique in software development used to control the release and availability of specific features. In machine learning, it can be used to:

- **Control Experimentation:** Enable or disable features dynamically to conduct A/B testing or incremental rollout of new features.
- **Facilitate Model Updates:** Implement and test new features without disrupting the main model, allowing for smoother transitions and easy rollback if issues arise.
- **Personalization:** Enable specific features for targeted user groups, useful in recommendation systems and personalized applications.

Feature flagging, when applied effectively, can significantly enhance the flexibility and adaptability of machine learning systems.

## 4.11 Perceptron Learning Algorithm

- Concept: *trials and errors*
- Task: Binary classification
- The foundation of complex algorithms and core of deep learning e.g. Neural networks

### 4.11.1 Problem Statement

Consider two labeled datasets, represented visually in Figure 4.30 on the left. The two classes in this example are represented by blue and red points. The objective is to construct a classifier that can predict the label of a new data point (illustrated as a grey triangle), given the labeled data of the two classes.

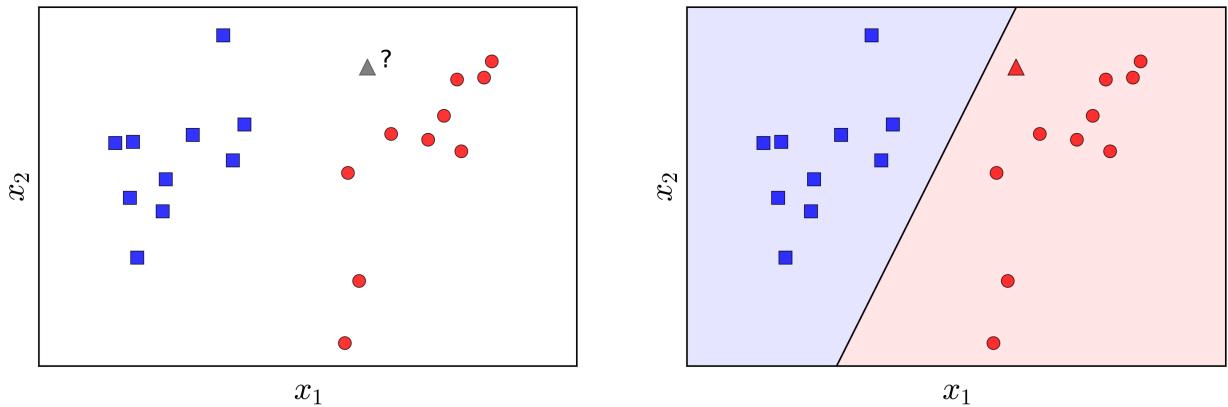


Figure 4.30: The Perceptron Problem

The Perceptron problem can be formally stated as follows: *Given two labeled classes, find a linear boundary such that all points belonging to class 1 lie on one side of the boundary, and all points belonging to class 2 lie on the opposite side. Assume that such a linear boundary exists.*

When such a boundary exists, the two classes are said to be *linearly separable*. Classification algorithms that construct linear boundaries are commonly referred to as *linear classifiers*.

Much like other iterative algorithms such as Gradient Descent, the basic idea behind PLA is to start with an initial guess for the boundary and iteratively improve it. At each iteration, the boundary is updated to move closer to an optimal solution. This update process is driven by minimizing a loss function that quantifies the misclassification error.

#### Notation

Let us denote the matrix of data points as  $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$ , where each column  $\mathbf{x}_i \in \mathbb{R}^{d \times 1}$  represents a data point in  $d$ -dimensional space. (Note: data points are represented as column vectors for convenience)

Assume that the corresponding labels are stored in a row vector  $\mathbf{y} = [y_1, y_2, \dots, y_N] \in \mathbb{R}^{1 \times N}$ , where  $y_i = 1$  if  $\mathbf{x}_i$  belongs to class 1 (blue), and  $y_i = -1$  if  $\mathbf{x}_i$  belongs to class 2 (red).

At any given point in the algorithm, suppose we have identified a linear boundary, which can be described by the equation:

$$f_{\mathbf{w}}(\mathbf{x}) = w_1x_1 + \dots + w_dx_d + w_0 = \mathbf{w}^T\bar{\mathbf{x}} = 0 \quad (4.18)$$

where  $\mathbf{w} \in \mathbb{R}^{d+1}$  is the weight vector, and  $\bar{\mathbf{x}} = [x_1, x_2, \dots, x_d, 1]^T \in \mathbb{R}^{d+1}$  is the augmented data point (including a bias term).

With  $\bar{\mathbf{x}}$ , we denote the extended data point by adding an element  $x_0 = 1$  to the beginning of the vector  $\mathbf{x}$ , similar to the method used in linear regression. Henceforth, we will assume that  $\mathbf{x}$  refers to the extended data point.

For simplicity, let us consider the case where each data point has two dimensions, i.e.,  $d = 2$ . Suppose the line equation  $w_1x_1 + w_2x_2 + w_0 = 0$  represents the solution we seek, as shown in Figure 4.31:

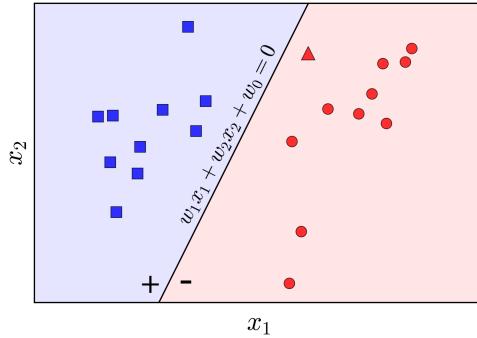


Figure 4.31: Equation of the decision boundary

We observe that points lying on the same side of this line will have the same sign for the function  $f_{\mathbf{w}}(\mathbf{x})$ . By adjusting the sign of  $\mathbf{w}$  as necessary, we assume that points on the positive side of the line (the blue-shaded half-plane) have positive labels (+), while points on the negative side (the red-shaded half-plane) have negative labels (-). These signs correspond to the label  $y$  for each class. Thus, if  $\mathbf{w}$  is a solution to the Perceptron problem, for a new unlabeled data point  $\mathbf{x}$ , we can determine its class by a simple computation:

$$\begin{aligned} \text{label}(\mathbf{x}) &= 1 \quad \text{if } \mathbf{w}^T\mathbf{x} \geq 0, \\ &= -1 \quad \text{otherwise} \end{aligned}$$

In summary:

$$\text{label}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T\mathbf{x})$$

where  $\text{sgn}$  denotes the sign function, and we assume that  $\text{sgn}(0) = 1$ .

### 4.11.2 Loss Function Construction

Next, we need to define a loss function for any given parameter  $\mathbf{w}$ . Still operating in two-dimensional space, assume the line  $w_1x_1 + w_2x_2 + w_0 = 0$  is given, as illustrated in Figure 4.32:

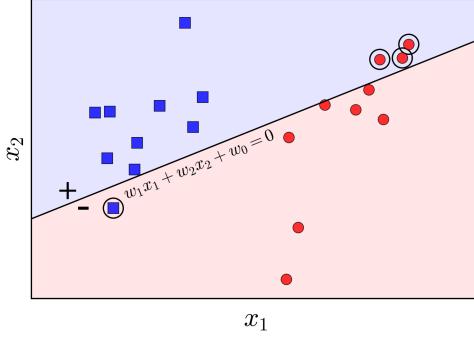


Figure 4.32: Arbitrary decision boundary with misclassified points circled.

In this case, the circled points represent the misclassified data points. Our goal is to have no misclassified points. The simplest loss function we might consider is one that counts the number of misclassified points and seeks to minimize this number:

$$\mathcal{J}_1(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \text{sgn}(\mathbf{w}^T \mathbf{x}_i))$$

where  $\mathcal{M}$  is the set of misclassified points (changes relatively to  $\mathbf{w}$ ). For each point  $\mathbf{x}_i \in \mathcal{M}$ , since the point is misclassified,  $y_i$  and  $\text{sgn}(\mathbf{w}^T \mathbf{x}_i)$  differ, and thus  $-y_i \text{sgn}(\mathbf{w}^T \mathbf{x}_i) = 1$ . Therefore,  $\mathcal{J}_1(\mathbf{w})$  is a count of the number of misclassified points. When this function reaches its minimum value of 0, no points are misclassified.

However, an important drawback of this function is that it is discrete and non-differentiable with respect to  $\mathbf{w}$ , making it challenging to optimize. We require a different loss function that is more amenable to optimization.

Consider the following loss function:  $\mathcal{J}(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \mathbf{w}^T \mathbf{x}_i)$

This function differs from  $\mathcal{J}_1()$  by omitting the sgn function. Note that for a misclassified point  $\mathbf{x}_i$ , the further it is from the boundary, the larger the value of  $-y_i \mathbf{w}^T \mathbf{x}_i$ , indicating a greater degree of error. The minimum value of this loss function is also 0, achieved when no points are misclassified. This loss function is considered superior to  $\mathcal{J}_1()$  because it penalizes points that are deeply misclassified more heavily, whereas  $\mathcal{J}_1()$  penalizes all misclassified points equally (with a value of 1), regardless of their proximity to the boundary.

At any given time, if we focus only on the misclassified points, the function  $\mathcal{J}(\mathbf{w})$  is differentiable. Consequently, we can employ optimization techniques such as Gradient Descent or Stochastic Gradient Descent (SGD) to minimize this loss function. Given the advantages of SGD in large-scale problems, we will follow this approach.

For a single misclassified data point  $\mathbf{x}_i$ , the loss function becomes:  $\mathcal{J}(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{w}^T \mathbf{x}_i$

The corresponding gradient is:  $\nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{x}_i$

Thus, the update rule is:

$$\mathbf{w} = \mathbf{w} + \eta y_i \mathbf{x}_i$$

where  $\eta$  is the learning rate, typically set to 1.

This results in a concise update rule:  $\mathbf{w}_{t+1} = \mathbf{w}_t + y_i \mathbf{x}_i$ . In other words, for each misclassified point  $\mathbf{x}_i$ , we multiply the point by its label  $y_i$  and add the result to  $\mathbf{w}$ , obtaining the updated weight vector.

We observe the following:

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = (\mathbf{w}_t + y_i \mathbf{x}_i)^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + y_i \|\mathbf{x}_i\|_2^2$$

If  $y_i = 1$ , since  $\mathbf{x}_i$  is misclassified,  $\mathbf{w}_t^T \mathbf{x}_i < 0$ . Additionally, since  $y_i = 1$ , we have  $y_i \|\mathbf{x}_i\|_2^2 = \|\mathbf{x}_i\|_2^2 \geq 1$  (note that  $x_0 = 1$ ), which implies that  $\mathbf{w}_{t+1}^T \mathbf{x}_i > \mathbf{w}_t^T \mathbf{x}_i$ . This means that  $\mathbf{w}_{t+1}$  moves toward correctly classifying  $\mathbf{x}_i$ . A similar argument applies when  $y_i = -1$ .

Thus, our intuition for this algorithm is as follows: pick a boundary, and for each misclassified point, move the boundary towards correctly classifying that point. Although previously correctly classified points may become misclassified during this process, the Perceptron Learning Algorithm (PLA) is guaranteed to converge after a finite number of steps (we will not elaborate further on the proof). In other words, we will eventually find a hyperplane that separates the two classes, provided that they are linearly separable.

### 4.11.3 Summary

The Perceptron Learning Algorithm can be summarized as follows:

1. Initialize the weight vector  $\mathbf{w}$  with values close to zero.
2. Randomly iterate through each data point  $\mathbf{x}_i$ :
  - If  $\mathbf{x}_i$  is correctly classified, i.e.,  $\text{sgn}(\mathbf{w}^T \mathbf{x}_i) = y_i$ , do nothing.
  - If  $\mathbf{x}_i$  is misclassified, update  $\mathbf{w}$  using the rule:  $\mathbf{w} = \mathbf{w} + \eta y_i \mathbf{x}_i$
3. Check how many points remain misclassified. If none, stop the algorithm. Otherwise, repeat from step 2.



# Chapter 5

## Support Vector Machine

### 5.1 Support Vector Machine

#### 5.1.1 Introduction

##### Distance from a point to a hyperplane

In two-dimensional space, we know that the distance from a point with coordinates  $(x_0, y_0)$  to a *line* with the equation  $w_1x + w_2y + b = 0$  is determined by:

$$\frac{|w_1x_0 + w_2y_0 + b|}{\sqrt{w_1^2 + w_2^2}}$$

In three-dimensional space, the distance from a point with coordinates  $(x_0, y_0, z_0)$  to a *plane* with the equation  $w_1x + w_2y + w_3z + b = 0$  is determined by:

$$\frac{|w_1x_0 + w_2y_0 + w_3z_0 + b|}{\sqrt{w_1^2 + w_2^2 + w_3^2}}$$

Moreover, if we drop the absolute value in the numerator, we can determine which side of the *line* or *plane* the point is on. Points that make the expression inside the absolute value positive lie on one side (which I call the *positive side* of the line), and points that make the expression negative lie on the other side (the *negative side*). Points lying on the *line/plane* will make the numerator equal to 0, meaning the distance is 0.

This can be generalized to higher-dimensional space: the distance from a point (vector) with coordinates  $\mathbf{x}_0$  to a *hyperplane* with the equation  $\mathbf{w}^T \mathbf{x} + b = 0$  is determined by:

$$\frac{|\mathbf{w}^T \mathbf{x}_0 + b|}{\|\mathbf{w}\|_2}$$

where  $\|\mathbf{w}\|_2 = \sqrt{\sum_{i=1}^d w_i^2}$ , with  $d$  being the number of dimensions in the space.

### Revisiting the Two-Class Classification Problem

We will revisit the problem from PLA. Assume there are two different classes represented by points in a high-dimensional space, and these two classes are *linearly separable*, meaning there exists a hyperplane that precisely separates the two classes. We want to find a hyperplane that separates these two classes, meaning all the points from one class are on the same side of the hyperplane and on the opposite side to the points from the other class. We already know that the PLA can achieve this, but it can give us many possible solutions, as illustrated in Figure 5.1 below:

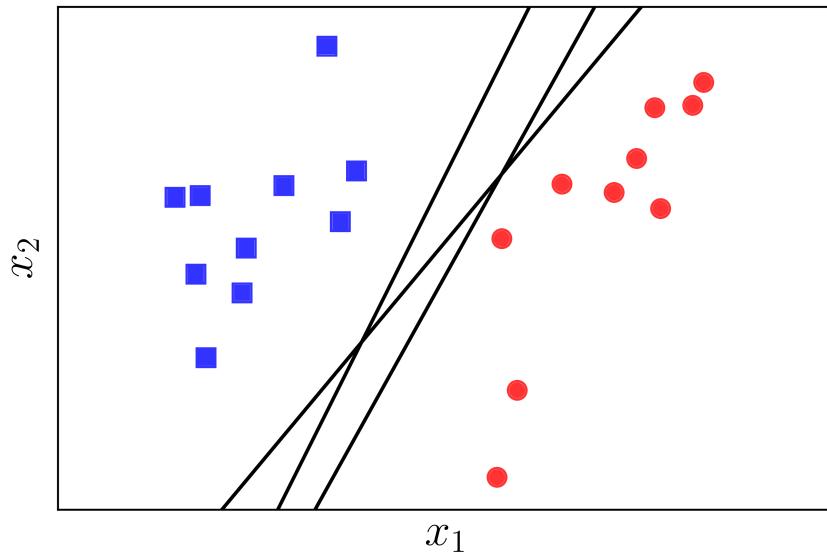


Figure 5.1: Separating hyperplanes for linearly separable classes.

**The question arises:** among the many possible hyperplanes, which one is the best *according to some criterion*? In the three lines shown in Figure 5.1 above, two of them are *biased* towards the red circular class. This might make the red class *unhappy*, as it feels that its *territory is being intruded upon too much*. Is there a way to find a hyperplane that both classes feel is the *fairest* and makes them *both happy*?

We need a criterion to measure the *happiness* of each class. Let's look at Figure 5.2 below:

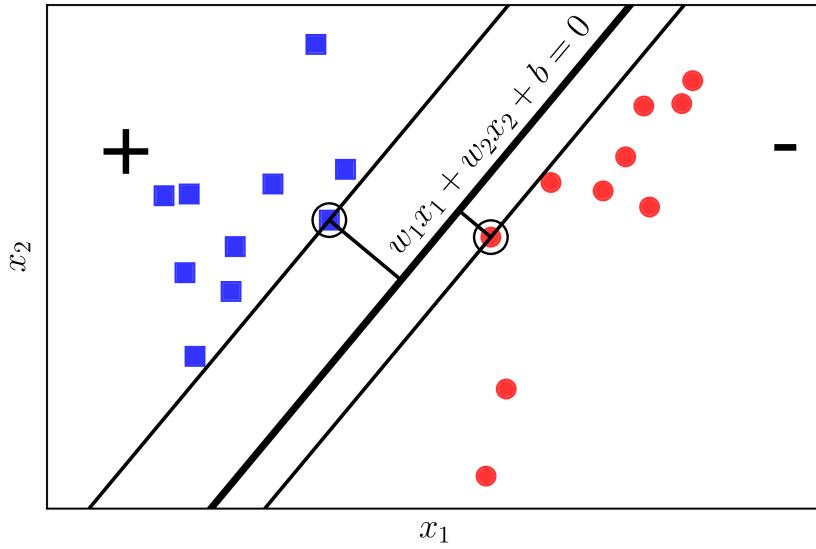


Figure 5.2: The margin of both classes is equal and as wide as possible.

If we define the *happiness* of a class to be proportional to the shortest distance from a point of that class to the separating hyperplane, then in Figure 5.2 (left), the red circular class is *not very happy* because the hyperplane is much closer to it compared to the blue square class. We need a hyperplane such that the distance from the nearest point of each class (the circled points) to the hyperplane is the same, so that it is *fair*. This equal distance is called the *margin*.

Having *fairness* is good, but we also need *equity*. *Fairness* where both classes are *equally unhappy* is not quite *ideal*.

Now let's consider Figure 5.2 (right), where the distance from the separating hyperplane to the closest points of each class is equal. Consider the two separating lines: the black solid line and the blue dashed line. Which one makes both classes *happier*? Clearly, it is the black solid line because it creates a wider *margin*.

A wider *margin* results in better classification because the separation between the two classes is more *clear-cut*. This, as you will see later, is one of the key reasons why *Support Vector Machine* yields better classification results compared to *Neural Networks with one layer*, such as the Perceptron Learning Algorithm.

The optimization problem in *Support Vector Machine* (SVM) is to find the hyperplane such that the *margin* is maximized. This is why SVM is also called the *Maximum Margin Classifier*.

### 5.1.2 Constructing the Optimization Problem for SVM

Assume the training set consists of pairs  $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)$ , where the vector  $\mathbf{x}_i \in \mathbb{R}^d$  represents the *input* of a data point, and  $y_i$  is the *label* of that data point.  $d$  is the number of dimensions in the data, and  $N$  is the number of data points. Assume that the label of each data point is either  $y_i = 1$  (class 1) or  $y_i = -1$  (class 2), as in PLA.

To help visualize, consider the two-dimensional case below. *The two-dimensional space is used for visualization purposes; the operations can be generalized to higher-dimensional spaces.*

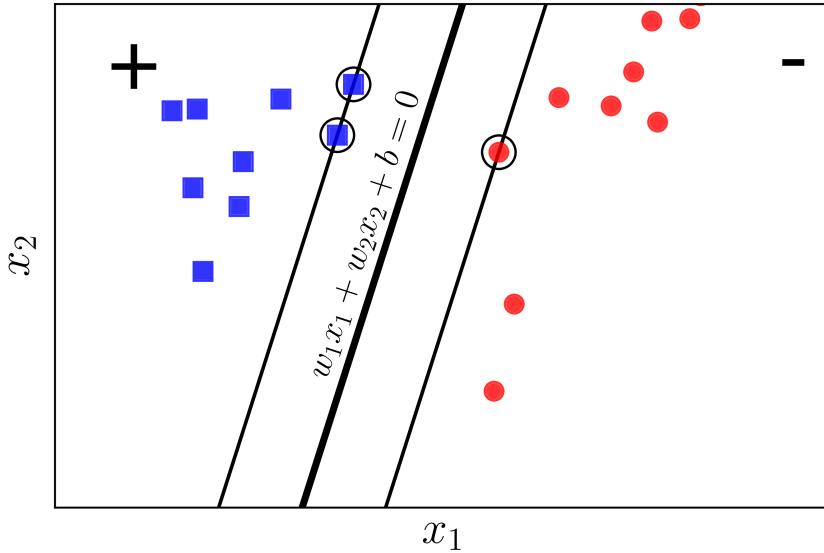


Figure 5.3: Analyzing the SVM problem

Assume that the blue square points belong to class 1, and the red circular points belong to class -1. The plane  $\mathbf{w}^T \mathbf{x} + b = w_1 x_1 + w_2 x_2 + b = 0$  is the separating hyperplane between the two classes. Furthermore, class 1 is on the *positive side*, and class -1 is on the *negative side* of the separating plane. If the opposite were true, we would just switch the sign of  $\mathbf{w}$  and  $b$ . Note that we need to determine the coefficients  $\mathbf{w}$  and  $b$ .

We observe the following important point: for any data pair  $(\mathbf{x}_n, y_n)$ , the distance from that point to the separating plane is: 
$$\frac{y_n(\mathbf{w}^T \mathbf{x}_n + b)}{\|\mathbf{w}\|_2}$$

This is evident because, as assumed earlier,  $y_n$  always has the same sign as the *side* of  $\mathbf{x}_n$ . Therefore,  $y_n$  has the same sign as  $(\mathbf{w}^T \mathbf{x}_n + b)$ , and the numerator is always a non-negative number.

With the separating hyperplane as above, the *margin* is the shortest distance from any point to the plane (regardless of which point in the two classes):

$$\text{margin} = \min_n \frac{y_n(\mathbf{w}^T \mathbf{x}_n + b)}{\|\mathbf{w}\|_2}$$

The optimization problem in SVM is to find  $\mathbf{w}$  and  $b$  such that this *margin* is maximized:

$$(\mathbf{w}, b) = \arg \max_{\mathbf{w}, b} \left\{ \min_n \frac{y_n(\mathbf{w}^T \mathbf{x}_n + b)}{\|\mathbf{w}\|_2} \right\} = \arg \max_{\mathbf{w}, b} \left\{ \frac{1}{\|\mathbf{w}\|_2} \min_n y_n(\mathbf{w}^T \mathbf{x}_n + b) \right\} \quad (5.1)$$

Solving this problem directly is very complex, but you will see that there is a way to simplify it.

The most important observation is that if we replace the coefficient vector  $\mathbf{w}$  with  $k\mathbf{w}$  and  $b$  with  $kb$ , where  $k$  is a positive constant, then the separating hyperplane does not change, meaning the distance from each point to the plane remains the same, hence the *margin* remains unchanged. Based on this property, we can assume:

$$y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1$$

for points closest to the separating plane, as shown in Figure 5.4 below:

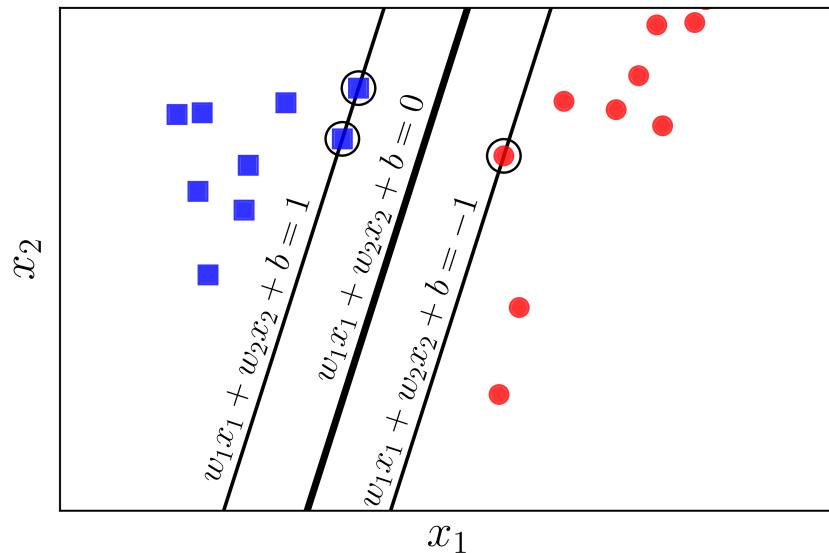


Figure 5.4: Points closest to the separating plane of the two classes are circled.

Thus, for all  $n$ :  $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1$

Therefore, the optimization problem (5.1) can be transformed into the following constrained

optimization problem:

$$\begin{aligned} (\mathbf{w}, b) &= \arg \max_{\mathbf{w}, b} \frac{1}{\|\mathbf{w}\|_2} \\ \text{subject to: } & y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1, \quad \forall n = 1, 2, \dots, N \end{aligned} \quad (5.2)$$

With a simple transformation, we can further reduce this to:

$$\begin{aligned} (\mathbf{w}, b) &= \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|_2^2 \\ \text{subject to: } & 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \quad \forall n = 1, 2, \dots, N \end{aligned} \quad (5.3)$$

Here, we take the reciprocal of the objective function, square it to make it differentiable, and multiply by  $\frac{1}{2}$  for a cleaner derivative expression.

#### **Important Observation:**

In problem (5.3), the objective function is a norm, which makes it convex. The inequality constraints are linear functions of  $\mathbf{w}$  and  $b$ , so they are also convex functions. Therefore, the optimization problem (5.3) has a convex objective function and convex constraints, making it a convex problem. Moreover, it is a Quadratic Programming problem. The objective function is also *strictly convex* because  $\|\mathbf{w}\|_2^2 = \mathbf{w}^T \mathbf{I} \mathbf{w}$ , and  $\mathbf{I}$  is the identity matrix—a positive definite matrix. From this, it can be concluded that the solution for SVM is *unique*.

At this point, this problem can be solved using tools for solving Quadratic Programming, such as CVXOPT library.

However, solving this problem becomes complicated when the dimensionality  $d$  of the data space and the number of data points  $N$  are large.

People often solve the dual problem of this problem. Firstly, the dual problem has interesting properties that make it more efficiently solvable. Secondly, during the formulation of the dual problem, it becomes evident that SVM can be applied to problems where the data is not *linearly separable*, meaning that the separating boundaries can be more complex than a simple plane.

**Classifying a New Data Point:** After finding the separating plane  $\mathbf{w}^T \mathbf{x} + b = 0$ , the class of any point is simply determined by:

$$\text{class}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T \mathbf{x} + b)$$

where the function `sgn` returns 1 if the argument is non-negative and -1 otherwise.

### 5.1.3 The Dual Problem for SVM. Origin of the name "Support Vector"

Recall that the optimization problem (5.3) is a convex problem. We know that if a [convex problem satisfies Slater's condition, then *strong duality* holds. And if *strong duality* holds,

then the solution of the problem is also the solution of the Karush-Kuhn-Tucker (KKT) conditions.

### Verifying Slater's Condition

Next, we will prove that the optimization problem (5.3) satisfies Slater's condition. Slater's condition states that if there exists  $\mathbf{w}, b$  such that:

$$1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) < 0, \quad \forall n = 1, 2, \dots, N$$

then *strong duality* holds.

This verification is relatively simple. Since we know that there is always a (hyper)plane that separates the two classes if they are *linearly separable*, meaning the problem has a solution, the *feasible set* of the optimization problem (5.3) must be non-empty. Therefore, there always exists a pair  $(\mathbf{w}_0, b_0)$  such that:

$$1 - y_n(\mathbf{w}_0^T \mathbf{x}_n + b_0) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (5.4)$$

$$\Leftrightarrow 2 - y_n(2\mathbf{w}_0^T \mathbf{x}_n + 2b_0) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (5.5)$$

Thus, simply choosing  $\mathbf{w}_1 = 2\mathbf{w}_0$  and  $b_1 = 2b_0$ , we get:

$$1 - y_n(\mathbf{w}_1^T \mathbf{x}_n + b_1) \leq -1 < 0, \quad \forall n = 1, 2, \dots, N$$

Hence, Slater's condition is satisfied.

### Lagrangian for the SVM Problem

The Lagrangian for the optimization problem (5.3) is:

$$\mathcal{L}(\mathbf{w}, b, \lambda) = \frac{1}{2} \|\mathbf{w}\|_2^2 + \sum_{n=1}^N \lambda_n (1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) \quad (5.6)$$

where  $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_N]^T$  and  $\lambda_n \geq 0, \forall n = 1, 2, \dots, N$ .

**Lagrange Dual Function** The Lagrange dual function is defined as:

$$g(\lambda) = \min_{\mathbf{w}, b} \mathcal{L}(\mathbf{w}, b, \lambda)$$

where  $\lambda \succeq 0$ .

The minimization of this function with respect to  $\mathbf{w}$  and  $b$  can be carried out by solving the system of partial derivative equations of  $\mathcal{L}(\mathbf{w}, b, \lambda)$  with respect to  $\mathbf{w}$  and  $b$ , setting them to zero:

$$\frac{\partial \mathcal{L}(\mathbf{w}, b, \lambda)}{\partial \mathbf{w}} = \mathbf{w} - \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n = 0 \Rightarrow \mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (5.7)$$

$$\frac{\partial \mathcal{L}(\mathbf{w}, b, \lambda)}{\partial b} = - \sum_{n=1}^N \lambda_n y_n = 0 \quad (5.8)$$

Substituting ((5.7)) and ((5.8)) back into ((5.6)), we obtain:

$$g(\lambda) = \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m$$

**This is the most important function in SVM.**

Consider the matrix:

$$\mathbf{V} = [y_1 \mathbf{x}_1, y_2 \mathbf{x}_2, \dots, y_N \mathbf{x}_N]$$

and the vector  $\mathbf{1} = [1, 1, \dots, 1]^T$ , we can rewrite  $g(\lambda)$  as:

$$g(\lambda) = -\frac{1}{2} \lambda^T \mathbf{V}^T \mathbf{V} \lambda + \mathbf{1}^T \lambda$$

Define  $\mathbf{K} = \mathbf{V}^T \mathbf{V}$ , and we have an important observation:  $\mathbf{K}$  is a positive semidefinite matrix. Indeed, for any vector  $\lambda$ :

$$\lambda^T \mathbf{K} \lambda = \lambda^T \mathbf{V}^T \mathbf{V} \lambda = \|\mathbf{V} \lambda\|_2^2 \geq 0$$

The above inequation is the definition of a positive semidefinite matrix.

Thus,  $g(\lambda) = -\frac{1}{2} \lambda^T \mathbf{K} \lambda + \mathbf{1}^T \lambda$  is a concave function.

### The Lagrange Dual Problem

From here, combining the Lagrange dual function and the constraints on  $\lambda$ , we obtain the Lagrange dual problem:

$$\begin{aligned} \lambda &= \arg \max_{\lambda} g(\lambda) \\ \text{subject to: } &\lambda \succeq 0 \\ &\sum_{n=1}^N \lambda_n y_n = 0 \end{aligned} \tag{5.9}$$

The second constraint is derived from (5.8). This is a convex problem because we are maximizing a concave objective function over a polyhedron. This problem is also a Quadratic Programming (QP) problem and can be solved using libraries like CVXOPT.

In this dual problem, the number of parameters to be found is  $N$ , which is the dimension of  $\lambda$ , i.e., the number of data points. Meanwhile, in the primal problem (5.3), the number of parameters to be found is  $d + 1$ , which is the total dimension of  $\mathbf{w}$  and  $b$ , i.e., the number of dimensions of each data point plus one. In many cases, the number of data points in the training set is much greater than the number of dimensions. If solved directly using Quadratic Programming tools, the dual problem can sometimes be more complex (and take more time) than the primal problem. However, the dual problem is particularly appealing

in the context of Kernel Support Vector Machine (Kernel SVM), which applies to problems where the data is not linearly separable or only nearly linearly separable. Additionally, based on the special properties of the KKT conditions, SVM can be solved using more efficient methods.

## KKT Conditions

Returning to the problem, since this is a convex problem with strong duality, the solution of the problem will satisfy the KKT conditions with variables  $\mathbf{w}, b$ , and  $\lambda$ :

$$1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (5.10)$$

$$\lambda_n \geq 0, \quad \forall n = 1, 2, \dots, N \quad (5.11)$$

$$\lambda_n(1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) = 0, \quad \forall n = 1, 2, \dots, N \quad (5.12)$$

$$\mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (5.13)$$

$$\sum_{n=1}^N \lambda_n y_n = 0 \quad (5.14)$$

Among these conditions, condition (5.12) is called the *complimentary slackness*, and is the most interesting. From it, we can immediately deduce that for any  $n$ , either  $\lambda_n = 0$  or  $1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) = 0$ . The latter case corresponds to:

$$\mathbf{w}^T \mathbf{x}_n + b = y_n \quad (5.15)$$

with the note that  $y_n^2 = 1, \forall n$ .

The points that satisfy (5.15) are the ones closest to the separating hyperplane, and they are the points circled in Figure 5.4 above. The two lines  $\mathbf{w}^T \mathbf{x}_n + b = \pm 1$  rest on the points that satisfy (5.15). Therefore, the points (vectors) that satisfy (5.15) are called *Support Vectors*. This is where the name *Support Vector Machine* comes from.

Another observation is that the number of points satisfying (5.15) is often very small compared to the  $N$  points. Using only these support vectors, we can determine the separating hyperplane. From another perspective, most  $\lambda_n$  are zero. Thus, although the vector  $\lambda \in \mathbb{R}^N$  can have a large number of dimensions, the number of nonzero components is very small. In other words, the vector  $\lambda$  is a *sparse* vector. Therefore, Support Vector Machine is also classified as a *Sparse Model*. Sparse Models often have more efficient (faster) solutions compared to similar models with dense solutions (most components nonzero). This is the second reason why the dual problem of SVM is often more interesting than the primal problem.

Continuing the analysis, for problems with a small number of data points  $N$ , the KKT conditions above can be solved by considering the cases where  $\lambda_n = 0$  or  $\lambda_n \neq 0$ . The total number of cases to consider is  $2^N$ . With  $N > 50$  (which is often the case), this number is very large, and solving this way becomes infeasible. I will not go deeper into how to solve the KKT system; in the next section, we will solve the optimization problem (5.9) using CVXOPT and the ‘sklearn’ library.

After finding  $\lambda$  from problem (5.9), we can determine  $\mathbf{w}$  using (5.13) and  $b$  using (5.12) and (5.14). Clearly, we only need to consider  $\lambda_n \neq 0$ .

Define the set  $\mathcal{S} = \{n : \lambda_n \neq 0\}$  and let  $N_{\mathcal{S}}$  be the number of elements in the set  $\mathcal{S}$ . For each  $n \in \mathcal{S}$ :

$$1 = y_n(\mathbf{w}^T \mathbf{x}_n + b) \Leftrightarrow b + \mathbf{w}^T \mathbf{x}_n = y_n$$

Although from just one pair  $(\mathbf{x}_n, y_n)$ , we can immediately determine  $b$  if  $\mathbf{w}$  is known, another version used for computing  $b$ , which is often more numerically stable, is

$$b = \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} (y_n - \mathbf{w}^T \mathbf{x}_n) = \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} \left( y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \quad (5.16)$$

which is the average over all calculations of  $b$ .

Previously, according to (5.13),  $\mathbf{w}$  was computed by

$$\mathbf{w} = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m \quad (5.17)$$

Important observation: To determine which class a new point  $\mathbf{x}$  belongs to, we need to determine the sign of the following expression:

$$\mathbf{w}^T \mathbf{x} + b = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x} + \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} \left( y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right)$$

This expression depends on computing the dot product between the vectors  $\mathbf{x}$  and each  $\mathbf{x}_n \in \mathcal{S}$ . This important observation will be useful in the Kernel SVM.

### 5.1.4 SVM in Python

#### Solve by formula

First, we import the necessary *modules* and create some synthetic data (this is the same data that I used in the figures above, so we know for sure that the two classes are *linearly separable*):

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
np.random.seed(42)

means = [[2, 2], [4, 2]]
cov = [[.3, .2], [.2, .3]]
N = 10
X0 = np.random.multivariate_normal(means[0], cov, N) # class 1
X1 = np.random.multivariate_normal(means[1], cov, N) # class -1
X = np.concatenate((X0.T, X1.T), axis = 1) # all data
y = np.concatenate((np.ones((1, N)), -1*np.ones((1, N))), axis = 1) # labels
```

Next, we solve the problem (5.9) using CVXOPT:

Listing 5.1: SVM simulation by formula in Python

```

from cvxopt import matrix, solvers
# build K
V = np.concatenate((X0.T, -X1.T), axis = 1)
K = matrix(V.T.dot(V)) # see definition of V, K near eq (8)

p = matrix(-np.ones((2*N, 1))) # all-one vector
# build A, b, G, h
G = matrix(-np.eye(2*N)) # for all lambda_n >= 0
h = matrix(np.zeros((2*N, 1)))
A = matrix(y) # the equality constraint is actually y^T lambda = 0
b = matrix(np.zeros((1, 1)))
solvers.options['show_progress'] = False
sol = solvers.qp(K, p, G, h, A, b)

l = np.array(sol['x'])
print('lambda = ')
print(l.T)

```

The result:

```

lambda =
[[ 8.54018321e-01   2.89132533e-10   1.37095535e+00   6.36030818e-10
  4.04317408e-10   8.82390106e-10   6.35001881e-10   5.49567576e-10
  8.33359230e-10   1.20982928e-10   6.86678649e-10   1.25039745e-10
  2.22497367e+00   4.05417905e-09   1.26763684e-10   1.99008949e-10
  2.13742578e-10   1.51537487e-10   3.75329509e-10   3.56161975e-10]]

```

We notice that most of the values of  $\lambda$  are very small, on the order of  $10^{-9}$  or  $10^{-10}$ . These are effectively zero due to computational inaccuracies. Only three values are non-zero, so we predict that there are three support vectors.

We now find the *support set*  $\mathcal{S}$  and solve the problem:

Listing 5.2: SVM with scikit-learn library

```

epsilon = 1e-6 # just a small number, greater than 1e-9
S = np.where(l > epsilon)[0]

VS = V[:, S]
XS = X[:, S]
yS = y[:, S]
lS = l[S]
# calculate w and b
w = VS.dot(lS)
b = np.mean(yS.T - w.T.dot(XS))

print('w = ', w.T)
print('b = ', b)

```

The result:

```
w = [[-2.00984381  0.64068336]]  
b = 4.66856063387
```

We illustrate the result:

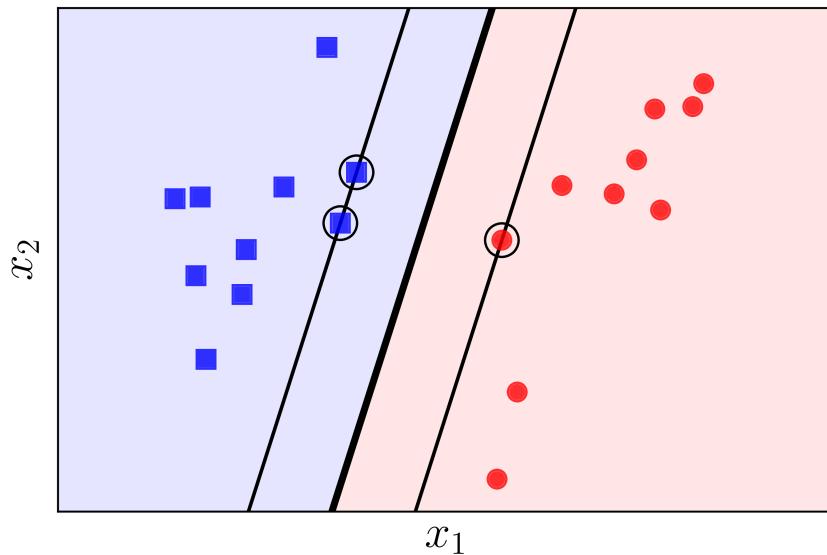


Figure 5.5: Illustration of the solution obtained by SVM.

The thick black line in the middle is the separating hyperplane found by SVM. It suggests that the calculations are most likely correct. To verify, we can also solve the problem using a standard library, such as `sklearn`.

### SVM by `sklearn.svm` library

We will now use the `sklearn.svm.SVC` built-in function.

Listing 5.3: SVM using built-in function

```
from sklearn.svm import SVC  
  
y1 = y.reshape((2*N,))  
X1 = X.T # each sample is one row  
clf = SVC(kernel = 'linear', C = 1e5) # just a big number  
  
clf.fit(X1, y1)  
  
w = clf.coef_  
b = clf.intercept_  
print('w = ', w)  
print('b = ', b)
```

The result:

```
w = [[-2.00971102  0.64194082]]  
b = [ 4.66595309]
```

### 5.1.5 Summary and Discussion

- For a binary classification problem where the two classes are *linearly separable*, there are infinitely many hyperplanes that can separate the classes. Each hyperplane defines a different classifier, and the distance from the closest point to the hyperplane is called the *margin*.
- The SVM problem seeks the hyperplane that maximizes the *margin*, ensuring that the data points are as far as possible from the separating hyperplane.
- The optimization problem in SVM is a convex problem with a *strictly convex* objective function, meaning the solution is unique. Additionally, the optimization problem is a Quadratic Programming (QP) problem.
- While we could directly solve the original SVM problem, it is typically solved using the *dual problem*, which is also a QP. The solution is *sparse*, making the dual problem more efficient to solve.
- For problems where the two classes are not *linearly separable*, we need to make sacrifices, or build another nonlinear kernel if the dataset has some special distributions (will elaborate afterwards). For n-ary classification problem, we need a Multiclass SVM (will not be elaborated as it is out-of-scope).

## 5.2 Soft Margin SVM

Consider two examples in Figure 5.6 below:

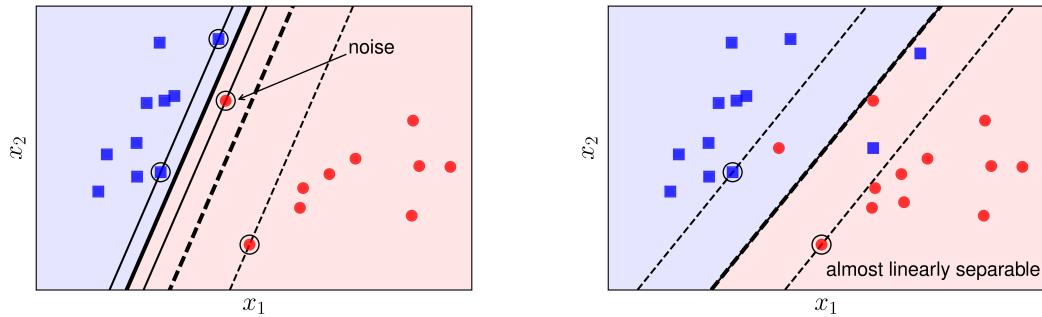


Figure 5.6: a: Noise within data; b: Almost linearly separable data

The default SVM will not work well (or even malfunction) in both cases:

- **Case 1:** The data is still *linearly separable*, but there is *noise* from the red circle class too close to the blue square class. In this case, if we use the *default SVM*, it will create a very small *margin*. Moreover, the decision boundary will be too close to the blue squares and far from the red circles. However, if we *sacrifice* this noise point, we can get a much better *margin* as illustrated by the dashed lines. Therefore, the *default SVM* is also considered *noise sensitive*.
- **Case 2:** The data is not *linearly separable* but is *nearly linearly separable* like in Figure 5.6b. In this case, if we use the *default SVM*, the optimization problem becomes *infeasible*, meaning the *feasible set* is empty, and the SVM optimization problem has no solution. However, if we *sacrifice* a few points near the boundary between the two classes, we can still create a good decision boundary, like the thick dashed line. The *support lines* formed by thin dashed lines still help create a large margin for this classifier. Each point that crosses over to the other side of the *support lines* (or *margin lines* or *boundary lines*) is considered to fall into the *unsafe region*. Note that the safe regions of the two classes are different, and they overlap in the area between the two support lines.

In both cases, the *margin* created by the decision boundary and the thin dashed lines is referred to as a *soft margin*. Similarly, the *default SVM* is also known as a *Hard Margin SVM*.

In this section, we will explore a variant of the *Hard Margin SVM* known as the *Soft Margin SVM*.

The optimization problem for the *Soft Margin SVM* can be approached in two different ways, both of which yield interesting results and can be extended into more complex and effective SVM algorithms.

**The first approach** is to solve a constrained optimization problem by solving the dual problem, similar to the *Hard Margin SVM*. The dual approach lays the foundation for the *Kernel SVM* for data that is truly not *linearly separable*.

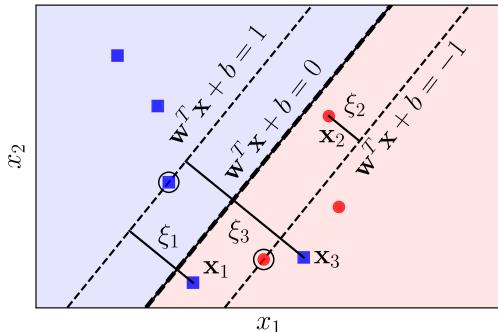
**The second approach** is to turn the problem into an *unconstrained* optimization problem. This problem can be solved using Gradient Descent methods. Therefore, this approach can be applied to large-scale problems. Additionally, in this solution, we will introduce a new loss function called the *hinge loss*. This loss function can be extended to the *multi-class classification* problem, which will be elaborate further on in the section of **Multiclass SVM**.

Let's first analyze the problem.

### 5.2.1 Mathematical Analysis

As mentioned above, to achieve a larger *margin* in *Soft Margin SVM*, we need to **sacrifice some data points by allowing them to fall into the unsafe region**. Of course, we must limit this *sacrifice*; otherwise, we could create a very large margin by *sacrificing* most points. Therefore, the objective function should be a combination that maximizes the *margin* and minimizes the *sacrifice*.

Like in the *Hard Margin SVM*, maximizing the *margin* can be reduced to minimizing  $\|\mathbf{w}\|_2^2$ . To define the *sacrifice*, let's refer to Figure 5.7 below:



Introducing slack variables  $\xi_n$ . For points within the *safe region*,  $\xi_n = 0$ . Points in the *unsafe region* but still on the correct side of the boundary correspond to  $0 < \xi_n < 1$ , for example,  $\mathbf{x}_2$ . Points on the wrong side of the boundary for their class correspond to  $\xi_n > 1$ , for example,  $\mathbf{x}_1$  and  $\mathbf{x}_3$ .

Figure 5.7

For each point  $\mathbf{x}_n$  in the entire training dataset, we *introduce* a slack variable  $\xi_n$  that measures the *sacrifice*. This variable is also known as the *slack variable*. For points  $\mathbf{x}_n$  within the *safe region*,  $\xi_n = 0$ . For points in the *unsafe region* like  $\mathbf{x}_1$ ,  $\mathbf{x}_2$ , or  $\mathbf{x}_3$ , we have  $\xi_n > 0$ .

If  $y_i = \pm 1$  is the *label* of  $\mathbf{x}_i$  in the *unsafe region*, then  $\xi_i = |\mathbf{w}^T \mathbf{x}_i + b - y_i|$ .

Let's recall the optimization problem for the *Hard Margin SVM*:

$$(\mathbf{w}, b) = \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|_2^2 \quad (5.18)$$

$$\text{subject to: } y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1, \forall n = 1, 2, \dots, N \quad (5.19)$$

For the *Soft Margin SVM*, the objective function will **include an additional term to minimize the sacrifice**. Therefore, the objective function becomes:

$$\frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n$$

where  $C$  is a positive constant, and  $\xi = [\xi_1, \xi_2, \dots, \xi_N]$ .

The constant  $C$  is used to adjust the balance between the *margin* and the sacrifice. This constant is pre-determined by the programmer or can be set via cross-validation.

The constraints will change slightly. For each data pair  $(\mathbf{x}_n, y_n)$ , instead of a *hard* constraint  $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1$ , we will have a *soft* constraint:

$$\begin{aligned} & y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n, \forall n = 1, 2, \dots, N \\ \Leftrightarrow & 1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \forall n = 1, 2, \dots, N \end{aligned}$$

And an additional constraint:  $\xi_n \geq 0, \forall n = 1, 2, \dots, N$ .

In summary, we have the optimization problem in standard form for *Soft-margin SVM*:

$$\begin{aligned} (\mathbf{w}, b, \xi) = & \arg \min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n \\ \text{subject to: } & 1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \forall n = 1, 2, \dots, N \\ & -\xi_n \leq 0, \forall n = 1, 2, \dots, N \end{aligned} \quad (5.20)$$

### Notes:

- When  $C$  is small, the sacrifice does not have much effect on the objective function. The algorithm will automatically adapt and minimize  $\|\mathbf{x}\|_2^2$ , or maximize *margin*, further increasing the sum  $\sum_{n=1}^N \xi_n$ .
- The optimization problem (5.20) has slack variables  $\xi_n$ . Data points with  $\xi_n = 0$  are in the safe region (satisfy constraint  $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1$ ), while data points with  $0 < \xi_n \leq 1$  are in the unsafe region (correctly classified but are too close to the hyperplane), and data points with  $\xi_n > 1$  are misclassified.
- The objective function is convex as it is the sum of two convex functions: the norm function and the linear function. The constraints are also convex. Therefore, this is a convex optimization problem, meaning that the local optimum is the global optimum.

Below, we will solve the optimization problem (5.20) in two different ways.

## 5.2.2 Lagrange Dual Problem

Note that this problem can be directly solved using QP toolboxes, but similar to the *Hard Margin SVM*, we will focus more on the dual problem.

First, we need to check the Slater's condition for the convex optimization problem (5.20). If this condition is satisfied, then *strong duality* will also hold, meaning the solution of the optimization problem (5.20) is the same as the solution of the KKT conditions.

### Checking Slater's Condition

It is clear that for all  $n = 1, 2, \dots, N$  and for all  $(\mathbf{w}, b)$ , we can always find positive values of  $\xi_n, n = 1, 2, \dots, N$  that are large enough so that:  $y_n(\mathbf{w}^T \mathbf{x}_n + b) + \xi_n > 1, \quad \forall n = 1, 2, \dots, N$

Therefore, this problem satisfies Slater's condition.

### Lagrangian of the Soft-margin SVM Problem

The Lagrangian for problem (5.20) is:

$$\mathcal{L}(\mathbf{w}, b, \xi, \lambda, \mu) = \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n + \sum_{n=1}^N \lambda_n (1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b)) - \sum_{n=1}^N \mu_n \xi_n$$

where  $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_N]^T \succeq 0$  and  $\mu = [\mu_1, \mu_2, \dots, \mu_N]^T \succeq 0$  are the Lagrange dual variables.

### Dual Problem

The dual function for the optimization problem (5.20) is:

$$g(\lambda, \mu) = \min_{\mathbf{w}, b, \xi} \mathcal{L}(\mathbf{w}, b, \xi, \lambda, \mu)$$

For each pair  $(\lambda, \mu)$ , we consider  $(\mathbf{w}, b, \xi)$  that satisfies the condition that the derivative of the Lagrangian is equal to 0:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0 \Leftrightarrow \mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (5.21)$$

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \Leftrightarrow \sum_{n=1}^N \lambda_n y_n = 0 \quad (5.22)$$

$$\frac{\partial \mathcal{L}}{\partial \xi_n} = 0 \Leftrightarrow \lambda_n = C - \mu_n \quad (5.23)$$

From (5.23), we can see that we only consider pairs  $(\lambda, \mu)$  such that  $\lambda_n = C - \mu_n$ . From here, we also have  $0 \leq \lambda_n, \mu_n \leq C, n = 1, 2, \dots, N$ . Substituting these expressions into the Lagrangian, we get the dual function:

$$g(\lambda, \mu) = \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m$$

Note that this function does not depend on  $\mu$ , but we need to consider the constraint (5.23). This constraint, along with the non-negativity condition on  $\lambda$ , can be rewritten as  $0 \leq \lambda_n \leq C$ , effectively reducing the variable  $\mu$ . Now, the dual problem is given by:

$$\begin{aligned} \lambda &= \arg \max_{\lambda} g(\lambda) \\ \text{subject to: } & \sum_{n=1}^N \lambda_n y_n = 0, \end{aligned} \quad (5.24)$$

$$0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N \quad (5.25)$$

This problem is similar to the [dual problem of the *Hard Margin SVM*, except that there is an upper bound on each  $\lambda_n$ . When  $C$  is very large, the two problems can be considered the same. The constraint (5.25) is also known as a *box constraint* because the feasible set of points  $\lambda$  satisfying this constraint forms a rectangular box in high-dimensional space.

This problem can also be solved entirely using standard QP solvers such as CVXOPT, as I did in the *Hard Margin SVM* example.

After finding  $\lambda$  from the dual problem, we still need to find the solution  $(\mathbf{w}, b, \xi)$  of the original problem. To do this, we need to examine the KKT conditions.

### KKT Conditions

The KKT conditions for the optimization problem of Soft Margin SVM are, for all  $n = 1, 2, \dots, N$ :

$$1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0 \quad (5.26)$$

$$-\xi_n \leq 0 \quad (5.27)$$

$$\lambda_n \geq 0 \quad (5.28)$$

$$\mu_n \geq 0 \quad (5.29)$$

$$\lambda_n(1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b)) = 0 \quad (5.30)$$

$$\mu_n \xi_n = 0 \quad (5.31)$$

$$\mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n$$

$$\sum_{n=1}^N \lambda_n y_n = 0$$

$$\lambda_n = C - \mu_n$$

(To make it easier to visualize, I have rewritten conditions (5.21), (5.22), (5.23) in this system.)

We can make some observations:

- If  $\lambda_n = 0$ , then from (5.23), we have  $\mu_n = C \neq 0$ . Combining this with (5.31), we get  $\xi_n = 0$ . In other words, no *sacrifice* occurs at  $\mathbf{x}_n$ , meaning  $\mathbf{x}_n$  lies in the safe region.
- If  $\lambda_n > 0$ , from (5.30), we have:  $y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1 - \xi_n$

- If  $0 < \lambda_n < C$ , from (5.23), we have  $\mu_n \neq 0$ , and from (14), we have  $\xi_n = 0$ . In other words,  $y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1$ , meaning these points lie *exactly* on the margin.
- If  $\lambda_n = C$ , then  $\mu_n = 0$ , and  $\xi_n$  can take any non-negative value. If  $\xi_n \leq 1$ ,  $\mathbf{x}_n$  will be correctly classified (still on the correct side of the decision boundary). Otherwise, for points with  $\xi_n > 1$ , they will be misclassified.
- $\lambda_n$  cannot be greater than  $C$  because then, according to (5.23),  $\mu_n < 0$ , which contradicts (5.29).

In addition, the points corresponding to  $0 < \lambda_n \leq C$  are now considered *support vectors*. Although these points may not lie on the *margins*, they are still considered support vectors because they contribute to the computation of  $\mathbf{w}$  through equation (4).

Thus, based on the values of  $\lambda_n$ , we can predict the relative position of  $\mathbf{x}_n$  with respect to the two margins. Define  $\mathcal{M} = \{n : 0 < \lambda_n < C\}$  and  $\mathcal{S} = \{m : 0 < \lambda_m \leq C\}$ . That is,  $\mathcal{M}$  is the set of indices of points that lie exactly on the *margins*—used to calculate  $b$ , and  $\mathcal{S}$  is the set of indices of the *support vectors*—used directly to calculate  $\mathbf{w}$ . Similar to the Hard Margin SVM, the parameters  $\mathbf{w}, b$  can be determined by:

$$\mathbf{w} = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m \quad (5.32)$$

$$b = \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} (y_n - \mathbf{w}^T \mathbf{x}_n) = \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left( y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \quad (5.33)$$

Note that the ultimate goal is to determine the label for a new point, not to compute  $\mathbf{w}$  and  $b$ , so we are more interested in determining the value of the following expression for any data point  $\mathbf{x}$ :

$$\mathbf{w}^T \mathbf{x} + b = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x} + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left( y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right)$$

In this calculation, we only need to consider the inner product between any two points.

## 5.3 Kernel SVM

### 5.3.1 Introduction

The soft margin approach solved the problem of imperfectly classified or noisy data; however, it is still limited to linear classification (with a decision boundary in the form  $Ax + b = 0$ ). For non-linearly separable data, the soft margin approach will also fail immediately.

The figure below shows the failure of the soft margin approach in a case of non-linearly separable data.

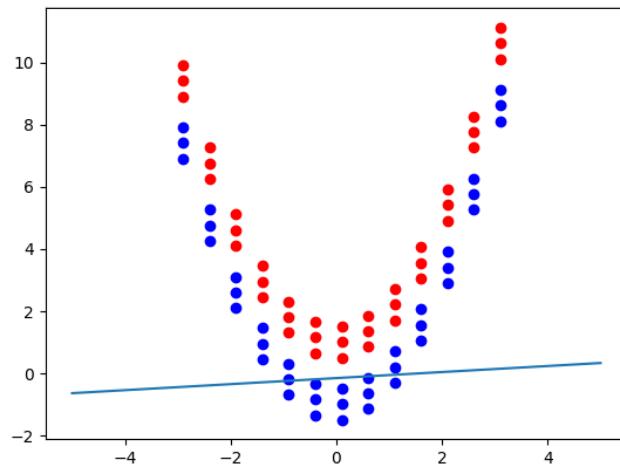


Figure 5.8: Failure of the soft margin approach

However, if we transform this data into a new space from  $(x_1, x_2)$  to  $(x_1^2, x_2)$ , the data transforms as follows:

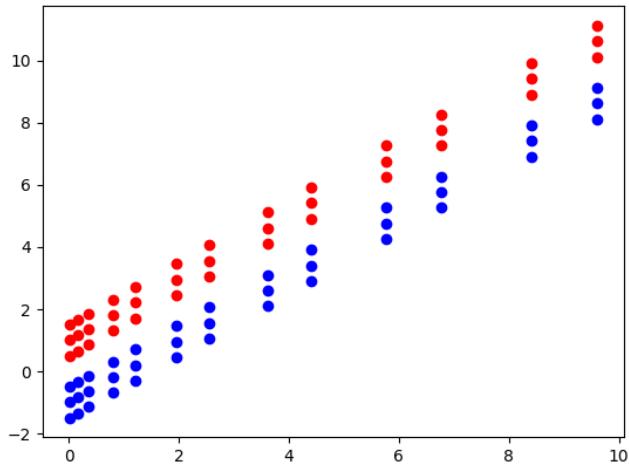


Figure 5.9: Data when transformed into a new space

We can see that the data can now be classified linearly. We only need to apply the soft margin approach or other linear classification methods to this transformed data.

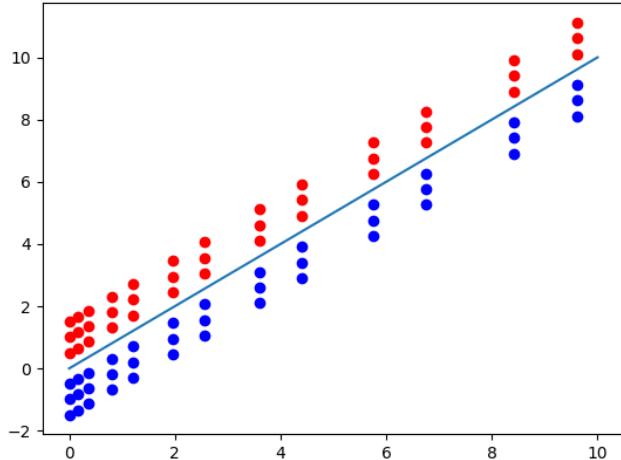


Figure 5.10: Classification after transforming data into the new space

The problem has been solved, but how do we know that transforming the space from  $(x_1, x_2)$  to  $(x_1^2, x_2)$  makes the data linearly separable? By plotting the data, we can observe that it can be separated by a parabolic curve, which is why we use that new space. However, for other problems where the data forms a complex curve or involves more than three dimensions, can we still manually determine the right space transformation? This approach of visualizing data is known as data visualization, a

technique currently being developed in data science. While there are visualization methods that can help us understand the data and find appropriate methods for solving problems, not all data can be visualized easily, making this a developing field. For complex data, visualizing and manually determining the transformation may not be feasible. Fortunately, there is another approach: the use of kernels.

With this idea, we transform the original space into a new one using the function  $\Phi(x)$ . In the example above,  $\Phi(x) = [x_1^2, x_2]^T$ .

In the new space, the problem becomes:

$$\begin{aligned}\lambda^* &= \arg \max_{\lambda} \sum_{i=1}^N \lambda_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \Phi(x_i)^T \Phi(x_j) \\ \text{subject to: } &\sum_{i=1}^N \lambda_i y_i = 0 \\ &0 \leq \lambda_i \leq C, \quad \forall i = 1, 2, \dots, N\end{aligned}$$

The decision function in the transformed space is:

$$\begin{aligned}f(\Phi(x)) &= w^T \Phi(x) + b \\ &= \sum_{m \in S} \lambda_m y_m \Phi(x_m)^T \Phi(x) + \frac{1}{N_M} \sum_{n \in M} \left( y_n - \sum_{m \in S} \lambda_m y_m \Phi(x_m)^T \Phi(x_n) \right)\end{aligned}$$

Calculating  $\Phi(x)$  directly is very difficult, and  $\Phi(x)$  may have a very large number of dimensions. Observing the above expressions, instead of calculating  $\Phi(x)$ , we only need to calculate  $\Phi(x)^T \Phi(z)$  for any two points  $x, z$ .

This technique is called the *kernel trick*. Methods that use this technique are known as *kernel methods*.

Define the kernel function as  $k(x, z) = \Phi(x)^T \Phi(z)$ . We can rewrite the problem as:

$$\begin{aligned}\lambda^* &= \arg \max_{\lambda} \sum_{i=1}^N \lambda_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j k(x_i, x_j) v \\ \text{subject to: } &\sum_{i=1}^N \lambda_i y_i = 0 \\ &0 \leq \lambda_i \leq C, \quad \forall i = 1, 2, \dots, N\end{aligned}$$

The decision function becomes:

$$f(\Phi(x)) = w^T \Phi(x) + b \tag{5.34}$$

$$= \sum_{m \in S} \lambda_m y_m k(x_m, x) + \frac{1}{N_M} \sum_{n \in M} \left( y_n - \sum_{m \in S} \lambda_m y_m k(x_m, x_n) \right) \tag{5.35}$$

### 5.3.2 Mathematical Analysis

Recall the dual problem for the soft margin SVM with data that is *almost linearly separable*:

$$\begin{aligned} \lambda &= \arg \max_{\lambda} \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m \\ \text{subject to: } & \sum_{n=1}^N \lambda_n y_n = 0 \\ & 0 \leq \lambda_n \leq C, \quad \forall n = 1, 2, \dots, N \end{aligned} \tag{5.36}$$

where:

- $N$ : Number of data pairs in the training set.
- $\mathbf{x}_n$ : Feature vector of the  $n^{th}$  data in the training set.
- $y_n$ : Label of the  $n^{th}$  data, equal to 1 or -1.
- $\lambda_n$ : Lagrange multiplier corresponding to the  $n^{th}$  data point.
- $C$ : A positive constant that balances the size of the margin and the compromise of data points in the unsafe region. When  $C = \infty$  or is very large, the soft margin SVM becomes a hard margin SVM.

After solving for  $\lambda$  in (5.36), the label of a new data point is determined by the sign of:

$$\sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x} + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left( y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right)$$

where:

- $\mathcal{M} = \{n : 0 < \lambda_n < C\}$  is the set of points on the margin.
- $\mathcal{S} = \{n : 0 < \lambda_n\}$  is the set of support points.
- $N_{\mathcal{M}}$  is the number of elements in  $\mathcal{M}$ .

With real-world data, it is difficult to have data that is *almost linearly separable*, so the solution to (5.36) may not produce a good classifier. Suppose we can find a function  $\Phi()$  such that after transforming to the new space, each data point  $\mathbf{x}$  becomes  $\Phi(\mathbf{x})$ , and in this new space, the data is *almost linearly separable*. In this case, the hope is that the solution to the soft margin SVM will provide a better classifier.

In the new space, (5.36) becomes:

$$\begin{aligned}\lambda &= \arg \max_{\lambda} \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \Phi(\mathbf{x}_n)^T \Phi(\mathbf{x}_m) \\ \text{subject to: } &\sum_{n=1}^N \lambda_n y_n = 0 \\ &0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N\end{aligned}\tag{5.37}$$

The label of a new data point is determined by:

$$\mathbf{w}^T \Phi(\mathbf{x}) + b = \sum_{m \in \mathcal{S}} \lambda_m y_m \Phi(\mathbf{x}_m)^T \Phi(\mathbf{x}) + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left( y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \Phi(\mathbf{x}_m)^T \Phi(\mathbf{x}_n) \right)\tag{5.38}$$

As mentioned above, calculating  $\Phi(\mathbf{x})$  directly for each data point may require a lot of memory and time because the dimensionality of  $\Phi(\mathbf{x})$  is often very large, possibly even infinite! Moreover, to determine the label of a new data point  $\mathbf{x}$ , we need to transform it to  $\Phi(\mathbf{x})$  in the new space and then take the dot product with all  $\Phi(\mathbf{x}_m)$ , where  $m$  is in the set of support points. To avoid this, we can use the following interesting observation.

In problem (5.37) and expression (5.38), we do not need to calculate  $\Phi(\mathbf{x})$  directly for every data point. We only need to calculate  $\Phi(\mathbf{x})^T \Phi(\mathbf{z})$  for any two data points  $\mathbf{x}, \mathbf{z}$ . This technique is known as the **kernel trick**. Methods that use this technique—calculating the dot product of two points in the new space instead of their individual coordinates—are collectively known as **kernel methods**.

Now, by defining the *kernel function*  $k(\mathbf{x}, \mathbf{z}) = \Phi(\mathbf{x})^T \Phi(\mathbf{z})$ , we can rewrite problem (5.37) and expression (5.38) as follows:

$$\begin{aligned}\lambda &= \arg \max_{\lambda} \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m k(\mathbf{x}_n, \mathbf{x}_m) \\ \text{subject to: } &\sum_{n=1}^N \lambda_n y_n = 0 \\ &0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N\end{aligned}\tag{5.39}$$

And:

$$\sum_{m \in \mathcal{S}} \lambda_m y_m k(\mathbf{x}_m, \mathbf{x}) + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left( y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m k(\mathbf{x}_m, \mathbf{x}_n) \right)$$

**Example:** Consider transforming a data point in two dimensions  $\mathbf{x} = [x_1, x_2]^T$  into a point in five dimensions  $\Phi(\mathbf{x}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2]^T$ . We have:

$$\begin{aligned}\Phi(\mathbf{x})^T \Phi(\mathbf{z}) &= [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2][1, \sqrt{2}z_1, \sqrt{2}z_2, z_1^2, \sqrt{2}z_1z_2, z_2^2]^T \\ &= 1 + 2x_1z_1 + 2x_2z_2 + x_1^2z_1^2 + 2x_1x_2z_1z_2 + x_2^2z_2^2 \\ &= (1 + x_1z_1 + x_2z_2)^2 = (1 + \mathbf{x}^T \mathbf{z})^2 = k(\mathbf{x}, \mathbf{z})\end{aligned}$$

In this example, it is clearly easier to calculate the kernel function  $k()$  for two data points than to compute each  $\Phi()$  and then multiply them together.

### 5.3.3 Properties of the Kernel Function

The kernel function must satisfy Mercer's theorem:

Given a set of data points  $x_1, \dots, x_n$  and any set of real numbers  $\lambda_1, \dots, \lambda_n$ , the kernel function  $K()$  must satisfy:

$$\sum_{i=1}^n \sum_{j=1}^n \lambda_i \lambda_j K(x_i, x_j) \geq 0$$

This means that the kernel function must be convex.

In practice, some kernel functions that do not strictly satisfy Mercer's theorem are still used because the results are acceptable.

### 5.3.4 Common Kernel Functions

In practice, the following kernel functions are commonly used:

- Linear:  $k(x, z) = x^T z$
- Polynomial:  $k(x, z) = (r + \lambda x^T z)^d$  where  $d$  is a positive integer indicating the degree of the polynomial.
- Radial Basis Function (RBF):  $k(x, z) = \exp(-\lambda \|x - z\|_2^2)$ ,  $\lambda > 0$
- Sigmoid:  $k(x, z) = \tanh(r + \lambda x^T z)$

In addition to the commonly used kernels above, there are many other kernels, such as string kernel, chi-square kernel, histogram intersection kernel, etc.

The effectiveness of different kernels is a topic of extensive research, but RBF (Gaussian kernel) is the most commonly used.

**Andrew Ng has a few tricks for choosing kernels:**

- Use a linear kernel (or logistic regression) when the number of features is larger than the number of observations (number of training examples).
- Use a Gaussian kernel when the number of observations is larger than the number of features.
- If the number of observations is larger than 50000, speed could be an issue when using the Gaussian kernel, so a linear kernel might be preferable.

Neural networks can handle all of these cases (and do well), but they may be slower. Another note: neural networks use convex optimization.

### 5.3.5 Summary

Neural Networks	SVM	General Characteristics
PLA	Hard Margin SVM	Two classes are <i>linearly separable</i>
Logistic Regression	Soft Margin SVM	Two classes are <i>almost linearly separable</i>
Softmax Regression	Multi-class SVM	Multi-class classification problem (linear boundaries)
Multi-layer Perceptron	Kernel SVM	Data is <i>not linearly separable</i>

# Chapter 6

## Ensemble Methods

### 6.1 Weak and Strong Learners

In machine learning, models are often classified as *weak learners* or *strong learners* based on their predictive performance and the role they play in ensemble methods.

#### 6.1.1 Weak Learners

A **weak learner** is a model that performs slightly better than random guessing. Typically, a weak learner has limited predictive power, and it may only be able to capture simple patterns in the data. Despite their simplicity, weak learners can be powerful components in ensemble learning, as they provide diverse perspectives that can be combined to form a more accurate model.

Examples of weak learners include:

- **Decision stumps:** A one-level decision tree that classifies based on a single feature.
- **Naive Bayes classifier:** Assumes feature independence, which simplifies calculations but limits accuracy for complex relationships.

Weak learners are commonly used in techniques like **Boosting**, where multiple weak models are sequentially trained, with each model focusing on correcting the errors of the previous model. Over multiple rounds, the ensemble of weak learners can achieve high accuracy, effectively transforming into a strong learner.

#### 6.1.2 Strong Learners

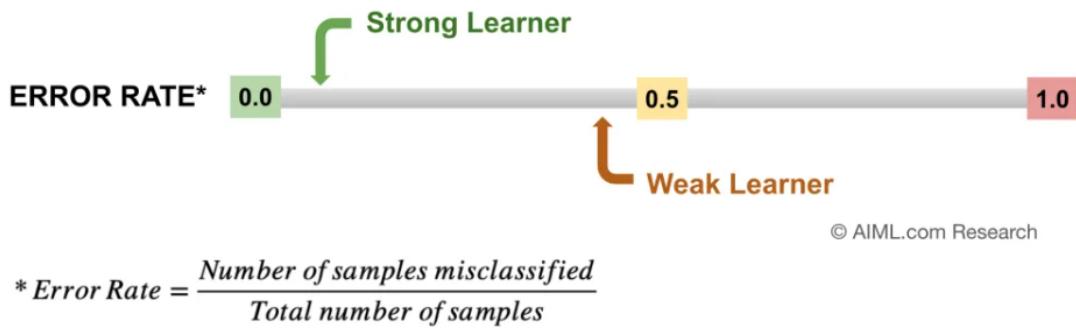
A **strong learner** is a model that has high predictive accuracy and can make reliable predictions on its own. Strong learners are capable of capturing complex patterns in data, making them useful for direct application in tasks requiring high accuracy.

Examples of strong learners include:

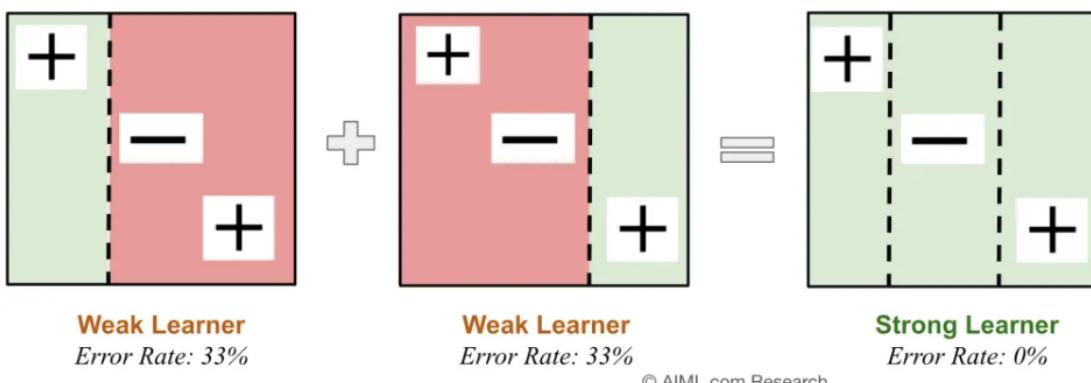
- **Deep neural networks:** With multiple layers and high-capacity architectures, they capture intricate relationships in data.
- **Random forests:** An ensemble of decision trees that reduces variance and provides robust predictions.

Strong learners are often used in **bagging** and **stacking** ensemble methods, where they are combined to enhance robustness and reduce the variance in predictions. Unlike weak learners, strong learners do not rely on iterative improvements and are effective as standalone models in many cases.

### 6.1.3 Weak vs. Strong Learners in Ensemble Learning



**Graphical representation of how weak learners can be combined to form a strong learner**



- Dotted lines represent the decision boundary for a weak learner (eg: a decision stump)
- + and - symbols represent the sample points belonging to a binary class classification problem

Figure 6.1: Weak and Strong Learners

- **Weak learners** are beneficial in Boosting techniques, where their individual limitations are compensated by focusing on different areas of error, ultimately creating a powerful ensemble model.
- **Strong learners** are typically used in bagging and stacking, where the aggregation of multiple strong learners results in a highly accurate and stable model.

The choice between weak and strong learners depends on the ensemble method and the problem complexity. Weak learners excel in adaptive methods like Boosting, while strong learners provide stable, high-accuracy ensembles when combined in parallel structures like bagging.

## 6.2 Ensemble Learning

### 6.2.1 History of Ensemble Learning

Ensemble learning has become a fundamental technique in machine learning, known for its ability to improve predictive performance by combining multiple models. The concept of ensembles was inspired by the idea of "crowd sourcing" for machines, where combining diverse predictions from multiple models reduces errors and enhances robustness. Early influential methods in ensemble learning included bagging, boosting, and random forests, which became widely used in predictive analytics and data science applications. Over time, ensemble methods like Random Forests and Gradient Boosting gained popularity for their effectiveness in reducing model variance and increasing stability. Pioneering work by scholars like Leo Breiman and Jerome Friedman established ensemble learning as a core strategy in modern machine learning, commonly applied in competitive modeling and real-world analytics.

### 6.2.2 Overview

In supervised learning, algorithms search a set of possible solutions, or hypothesis space, to identify a hypothesis that makes accurate predictions for a given problem. Although this space may include excellent hypotheses, locating one can be challenging. **Ensemble learning** addresses this by combining multiple hypotheses, potentially improving predictive performance.

**Ensemble learning** involves training multiple machine learning algorithms to work together on a classification or regression task. These individual algorithms, known as "base models," "base learners," or "weak learners," can come from the same or different modeling techniques. The goal is to train a diverse set of weak models on the same task, each with limited predictive accuracy (high bias) but varying prediction patterns (high variance). By combining weak learners — models that alone are not highly accurate — ensemble learning creates a single model with greater accuracy and lower variance.

Ensemble learning typically employs three main strategies: *bagging*, *boosting*, and *stacking*.

- **Bagging (Bootstrap Aggregating)**: Focuses on model diversity by training multiple versions of the same model on different random samples from the training data, leading to a homogeneous parallel ensemble.
- **Boosting**: Trains models sequentially, with each model addressing errors made by the previous one. This forms an additive model that aims to reduce overall error, known as a sequential ensemble.
- **Stacking (or Blending)**: Combines independently trained, diverse models into a final model, creating a heterogeneous parallel ensemble. Models in stacking are chosen based on the specific task, like pairing clustering methods with other models.

Common ensemble applications include *random forests* (an extension of bagging), *boosted tree models*, and *gradient-boosted trees*.

Ensemble learning also relates to multiple classifier systems, which include hybrid models using different types of base learners. Evaluating an ensemble's predictions can require more computation than using a single model. However, an ensemble approach may improve accuracy more efficiently than scaling up a single model, balancing computational resources for better performance. Fast algorithms like decision trees are often used in ensembles (e.g., random forests), but even slower algorithms can benefit from ensemble techniques.

**Example:** In your AI-generated text detection project, ensemble learning can improve accuracy by combining different classifiers focusing on unique aspects of text, such as lexical, stylistic, and semantic patterns. Here's an example with ensemble strategies:

- **Bagging:** Train several versions of each classifier on random samples to reduce overfitting.
- **Boosting:** Sequentially train models to focus on errors from previous ones, helping detect subtle AI patterns.
- **Stacking:** Use a meta-classifier to combine outputs from each model, leveraging their strengths for higher accuracy.

### 6.2.3 Types of Ensemble

#### 6.2.3.1 Bagging

**Description:** Bagging (Bootstrap Aggregating) creates multiple versions of the same model by training on different random samples from the training data, reducing variance and avoiding overfitting. Each model (often decision trees) is trained independently, and their predictions are averaged or voted on for a final decision.

**Use Cases:** Bagging is widely used in random forests, where it combines multiple decision trees for tasks like classification and regression.

**AI-Generated Text Detection Project:** Yes, bagging can be used here by training multiple classifiers on different subsets of text data. This can help capture various linguistic features present in AI-generated vs. human-generated text, making the model more robust to different writing styles.

#### 6.2.3.2 Boosting

**Description:** Boosting is a sequential ensemble method where each model is trained on the errors of the previous one, thus focusing more on difficult cases. Boosting combines weak learners to create a strong classifier, gradually reducing bias and improving accuracy.

**Use Cases:** Boosting is effective in applications requiring high accuracy, like image recognition and fraud detection, and is popular in models like AdaBoost and Gradient Boosting.

**AI-Generated Text Detection Project:** Yes, boosting can improve the detection model by focusing on misclassified examples. For instance, if certain types of AI-generated text are harder to classify, boosting helps by training subsequent models on those specific examples, refining the model's accuracy.

### 6.2.3.3 Stacking

**Description:** Stacking, or stacked generalization, involves training multiple base models and using a meta-model to learn from their combined predictions. The meta-model synthesizes the outputs of the base models for improved performance.

**Use Cases:** Stacking is useful in complex tasks like recommendation systems and predictive modeling in finance, where combining diverse models increases robustness.

**AI-Generated Text Detection Project:** Yes, stacking can be beneficial by combining models trained on different linguistic features (e.g., syntactic, stylistic, and semantic classifiers) to improve detection performance. The meta-model can capture interactions among these features for better classification.

### 6.2.3.4 Voting

**Description:** Voting combines the predictions of multiple models, with the final prediction based on the majority vote (for classification) or average (for regression). This method helps reduce individual model bias.

**Use Cases:** Voting is often used when there are different models available that perform well individually, such as in spam detection and sentiment analysis.

**AI-Generated Text Detection Project:** Yes, voting can be applied here by combining classifiers trained on different features of the text. Majority voting among these classifiers can improve the overall reliability of the detection model.

### 6.2.3.5 Bayes Optimal Classifier

#### Description:

The Bayes optimal classifier is a classification technique that represents the theoretical best possible model by combining all hypotheses in the hypothesis space, weighted by their posterior probability. On average, no other ensemble can outperform it. The Naive Bayes classifier is a feasible version that assumes conditional independence of the data given the class, simplifying computation.

In the Bayes optimal classifier, each hypothesis is given a vote proportional to the likelihood that the training dataset would have been generated if that hypothesis were true. This vote is also weighted by the prior probability of the hypothesis.

The Bayes optimal classifier can be expressed with the following equation:

$$y = \arg \max_{c_j \in C} \sum_{h_i \in H} P(c_j|h_i)P(T|h_i)P(h_i)$$

where  $y$  is the predicted class,  $C$  is the set of all possible classes,  $H$  is the hypothesis space,  $P$  represents a probability, and  $T$  is the training data. As an ensemble, the Bayes optimal classifier may represent a hypothesis not necessarily within  $H$ , but rather in the ensemble space — the space of all possible combinations of hypotheses in  $H$ .

This formula can also be derived using Bayes' theorem, which states that the posterior is proportional to the likelihood times the prior:

$$P(h_i|T) \propto P(T|h_i)P(h_i)$$

Thus, the classifier can also be expressed as:

$$y = \arg \max_{c_j \in C} \sum_{h_i \in H} P(c_j|h_i)P(h_i|T)$$

**Use Cases:** This approach is mostly theoretical, serving as a benchmark in decision theory and optimal classifier research.

**AI-Generated Text Detection Project:** No, it's impractical for real-world text detection due to computational demands. However, this concept can be applied in smaller-scale, highly specific classification problems where computational resources are available.

#### 6.2.3.6 Bayesian Model Averaging

**Description:** Bayesian Model Averaging (BMA) makes predictions by averaging over models weighted by their posterior probabilities, providing robustness to model uncertainty. This method often yields better predictions than single models, especially when multiple models perform similarly on the training set but generalize differently. BMA relies on choosing a prior probability for each model, with BIC and AIC as common choices, each reflecting different preferences for model complexity.

**BIC (Bayesian Information Criterion):** BIC is a criterion used to select models based on goodness of fit while penalizing model complexity more strongly than AIC. It is calculated as:

$$\text{BIC} = k \ln(n) - 2 \ln(L)$$

where  $k$  is the number of parameters in the model,  $n$  is the number of data points, and  $L$  is the likelihood of the model given the data. BIC tends to favor simpler models, particularly as sample size  $n$  increases, which helps avoid overfitting.

**AIC (Akaike Information Criterion):** AIC balances the trade-off between model fit and complexity with a lower penalty for additional parameters compared to BIC, making it less conservative. It is calculated as:

$$\text{AIC} = 2k - 2 \ln(L)$$

where  $k$  is the number of parameters and  $L$  is the model's likelihood given the data. Models with lower AIC values are generally preferred, as they achieve a balance between fit and simplicity without over-penalizing complexity.

In BMA, the choice between BIC and AIC affects how models are weighted in the averaging process, with models that have lower AIC or BIC scores (depending on the criterion chosen) receiving higher weights. This weighting approach helps to improve the reliability of predictions by favoring models that provide a balance of accuracy and simplicity.

**Use Cases:** BMA is applied in forecasting, model selection, and tasks where uncertainty quantification is critical, such as climate modeling and environmental predictions.

**AI-Generated Text Detection Project:** No, BMA might not be ideal for text detection due to its computational complexity and focus on uncertainty quantification. Instead, it is better suited for predictive tasks that require robust uncertainty estimation, like medical outcome prediction.

#### 6.2.3.7 Bayesian Model Combination

**Description:** Bayesian Model Combination (BMC) is an algorithmic improvement over Bayesian Model Averaging (BMA). Instead of sampling each model individually, BMC samples from the space of possible ensembles with model weights drawn from a Dirichlet distribution. This approach mitigates BMA's tendency to heavily favor a single model, yielding a more balanced combination and better average results. BMC is computationally more demanding than BMA but provides improved accuracy by finding an optimal weighting of models closer to the data distribution.

**Use Cases:** BMC is valuable in fields requiring robust probabilistic modeling, such as natural language processing and medical diagnosis, where it's crucial to quantify uncertainty in predictions.

**AI-Generated Text Detection Project:** Yes, BMC can be applied to provide probabilistic outputs, which helps the detection model indicate the likelihood of text being AI-generated versus human-written, enhancing interpretability and model confidence.

#### 6.2.3.8 Amended Cross-Entropy Cost

##### Description:

Cross-Entropy is a common cost function used in classification tasks, particularly to measure the difference between the true probability distribution  $p$  and the predicted probability distribution  $q$  from a model. The Cross-Entropy cost function is defined as:

$$H(p, q) = - \sum_i p(i) \log q(i)$$

where  $p(i)$  is the true probability of class  $i$ , and  $q(i)$  is the predicted probability of class  $i$ .

This approach can be modified to encourage diversity among base classifiers in an ensemble, leading to better generalization. The Amended Cross-Entropy Cost is defined as:

$$e^k = H(p, q^k) - \frac{\lambda}{K} \sum_{j \neq k} H(q^j, q^k)$$

where  $e^k$  is the cost function of the  $k^{\text{th}}$  classifier,  $q^k$  is the probability of the  $k^{\text{th}}$  classifier,  $p$  is the true probability that we need to estimate, and  $\lambda$  is a parameter between 0 and 1 that defines the desired diversity level. When  $\lambda = 0$ , each classifier aims to optimize individually, while  $\lambda = 1$  encourages maximum diversity within the ensemble.

**Use Cases:** Useful in classification tasks requiring robust generalization, such as image and speech recognition.

**AI-Generated Text Detection Project:** Yes, this could be beneficial by ensuring diversity among classifiers focusing on different textual features. This can enhance generalization to different types of AI-generated texts, improving detection accuracy.

## 6.3 Voting

**Ensemble Voting** is a widely-used ensemble learning technique where multiple models (or "base learners") independently make predictions, and a final decision is made by combining these predictions. This approach enhances model accuracy, robustness, and interpretability by leveraging the strengths of each base learner.

### 6.3.1 Types of Ensemble Voting Methods

There are several types of ensemble voting techniques commonly used in machine learning:

- **Majority (Hard) Voting:** Each model in the ensemble makes a categorical prediction (class label), and the class label receiving the majority of votes is chosen as the final prediction. Majority voting is particularly effective when individual models are accurate and diverse. In cases where votes are tied, tie-breaking rules (e.g., selecting the class with the highest confidence) can be applied.
- **Weighted Voting:** In this method, different weights are assigned to each model's prediction based on its performance or reliability. Each model's prediction is multiplied by its corresponding weight, and the final decision is determined by summing or averaging these weighted predictions. Weights can be manually assigned or optimized through techniques like cross-validation.
- **Soft Voting:** Soft voting is used when models provide probability estimates or confidence scores for each class label instead of discrete predictions. The predicted probabilities from each model are averaged, and the class with the highest average probability is selected as the final prediction. This approach allows models to incorporate their confidence levels in the final decision.
- **Voting Regressor:** Ensemble voting can also be applied to regression tasks, where each base model outputs a numerical prediction. In this case, the final output is typically the average of all predictions, which helps reduce variance and improves stability.

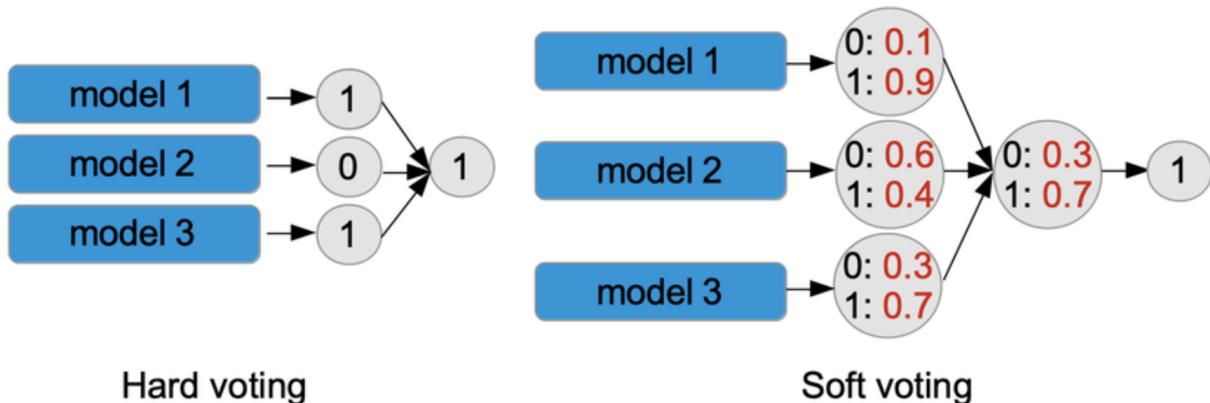


Figure 6.2: Soft and Hard Voting Examples

### 6.3.2 Theoretical Advantages of Ensemble Voting

Ensemble voting provides several advantages over individual models:

- **Improved Accuracy:** By combining the predictions of multiple models, ensemble voting often achieves higher accuracy than any single model, as it reduces both bias and variance.
- **Robustness:** Voting makes the model more robust to errors or noise in individual models. It mitigates the impact of inaccurate predictions from any single model by considering multiple viewpoints.
- **Model Diversity and Complementary Strengths:** Ensemble voting allows for the combination of various types of models, each with unique strengths and weaknesses. This diversity enables the ensemble to capture complementary patterns and features in the data, enhancing overall performance.
- **Interpretability:** Voting provides insights into the importance and agreement among different models. By analyzing the consistency of predictions, ensemble voting aids in understanding the relative contributions of each model, making it easier to interpret the ensemble's decision-making process.

### 6.3.3 Applications of Ensemble Voting in Machine Learning

Ensemble voting is widely used in practice, particularly in fields like *finance, healthcare, and computer vision*. Applications include:

- **Voting Classifier:** Commonly used in classification tasks, such as fraud detection and medical diagnosis, where combining multiple classifiers improves prediction accuracy.

- **Voting Regressor:** Useful in regression tasks, such as predictive analytics in finance or weather forecasting, where averaging predictions from multiple regressors reduces prediction error.
- **Random Forests and Bagging:** Random Forests and bagging methods rely on voting mechanisms (usually hard voting) across multiple decision trees to achieve robust classification and regression performance.

#### 6.3.4 Assumptions and Limitations of Ensemble Voting

While ensemble voting provides substantial benefits, there are important considerations:

- **Model Independence and Diversity:** Ensemble voting assumes that base models are independent and diverse in terms of algorithm, hyperparameters, training data, or feature representations. Greater diversity among models enhances accuracy and robustness.
- **Computational Complexity:** Ensemble voting can be computationally expensive due to the need to train and maintain multiple models, which may require additional resources for storage and processing.
- **Correlated Models:** If the base models are highly correlated (i.e., make similar errors), the ensemble's effectiveness diminishes, as the benefits of diversity are reduced.

**Conclusion** Ensemble voting is a powerful technique in machine learning that combines multiple models to improve prediction accuracy, robustness, and interpretability. Different types of voting (e.g., majority voting, weighted voting, and soft voting) cater to different needs, and the choice depends on the problem's requirements. With applications across various domains, ensemble voting remains a versatile tool for enhancing machine learning models.

## 6.4 Bagging

### 6.4.1 History of Ensemble Bagging

Bagging, short for "Bootstrap Aggregating," was introduced by Leo Breiman in the 1990s as a method to reduce the variance of decision trees. The technique involves creating multiple bootstrap samples (random samples with replacement) from the training data, training a model on each sample, and averaging their predictions. Bagging proved particularly useful for unstable learners like decision trees, which are sensitive to data fluctuations. Breiman's Random Forests further expanded on bagging by combining bootstrap sampling with random feature selection, leading to a powerful ensemble model known for its accuracy and robustness. Bagging remains a cornerstone of ensemble methods, enhancing the stability of machine learning models.

**Description** Bagging, is an ensemble meta-algorithm in machine learning used to enhance the stability and accuracy of models, particularly for tasks in classification and regression. By reducing variance, bagging effectively mitigates overfitting, making it especially useful for high-variance algorithms like decision trees. Though typically used with decision trees, bagging can apply to any model, making it a versatile approach in model averaging.

### 6.4.2 Technique

Given a standard training set  $D$  of size  $n$ , bagging generates  $m$  new training sets  $D_i$ , each of size  $n'$ , by sampling from  $D$  uniformly with replacement. Sampling with replacement creates **bootstrap samples**, where each  $D_i$  may contain repeated observations. When  $n' = n$  and  $n$  is large, each bootstrap sample  $D_i$  will, on average, contain around  $(1 - 1/e) \approx 63.2\%$  unique examples from  $D$ , with the remainder being duplicates.

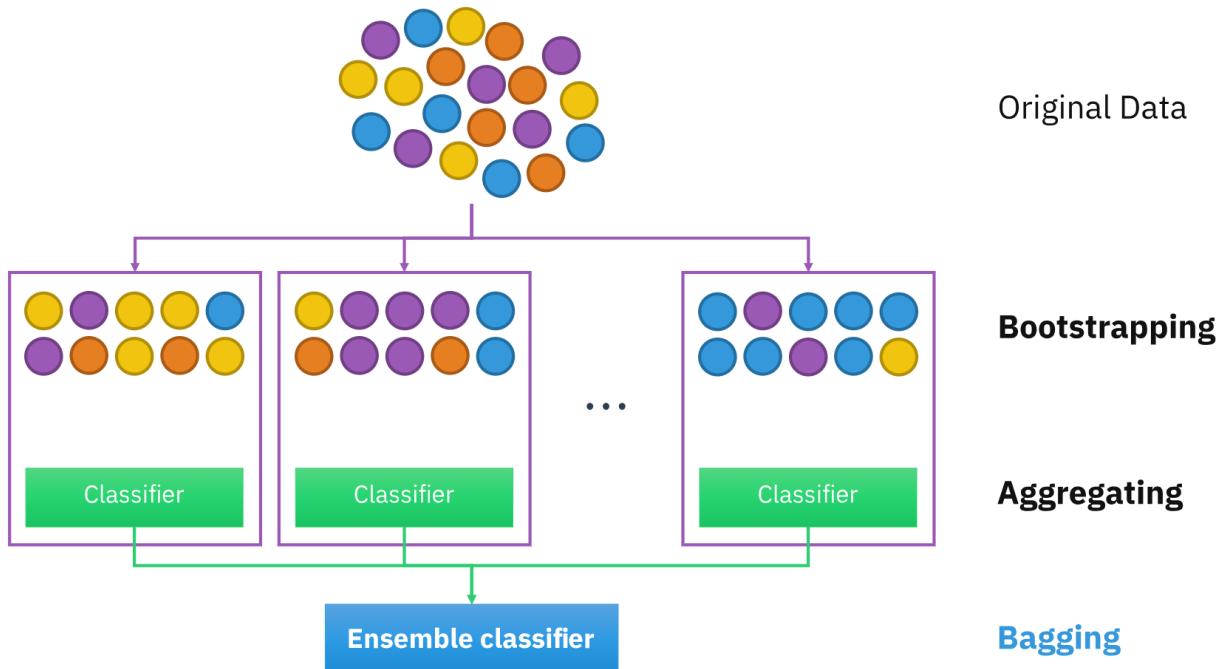


Figure 6.3: An illustration for the concept of bootstrap aggregating

After creating the  $m$  bootstrap samples, each sample  $D_i$  is used to train a separate model  $M_i$ . For prediction, the bagging ensemble combines the output of each model by averaging (in regression) or majority voting (in classification), producing a final, aggregated prediction.

### 6.4.3 Datasets in Bagging

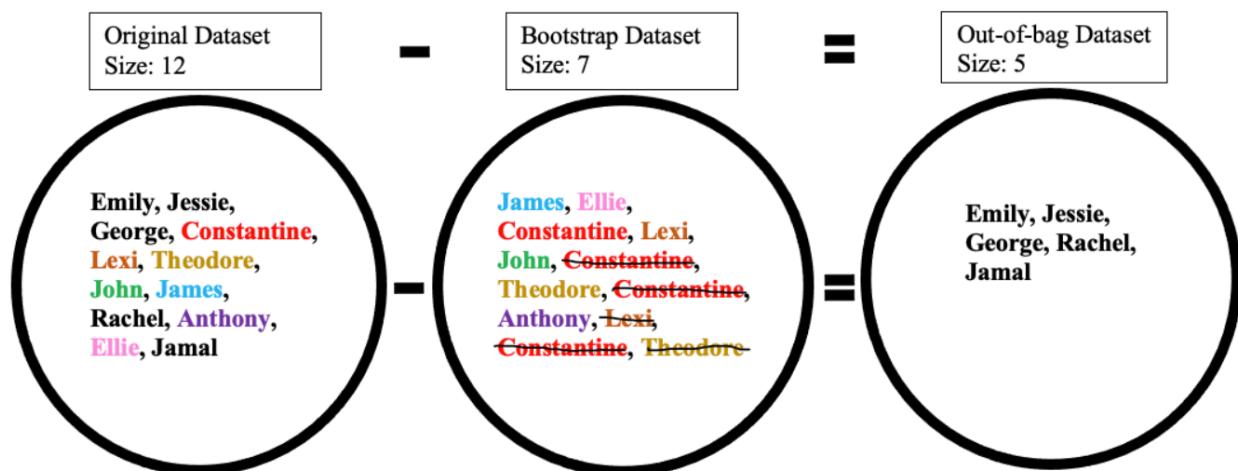


Figure 6.4: Bootstrap Aggregating

Bagging involves three main datasets:

- *Original Dataset*: The initial dataset, containing all samples.
- *Bootstrap Dataset*: Created by sampling with replacement from the original dataset, containing some duplicates, and it **has the same size as the original dataset**.
- *Out-of-Bag (OOB) Dataset*: The samples left out during bootstrapping, which can be used to assess model accuracy.

#### 6.4.4 Classification Algorithm Process using Bootstrap Sampling

For classification tasks, we can use a bootstrap-based ensemble algorithm, which combines predictions from multiple classifiers. Given a training set  $D$ , an inducer  $I$ , and the number of bootstrap samples  $m$ , the following steps outline the process to generate a final classifier  $C^*$ :

1. **Generate Bootstrap Samples**: Create  $m$  new training sets  $D_i$  by sampling from  $D$  with replacement.
2. **Train Individual Classifiers**: For each bootstrap sample  $D_i$ , train a classifier  $C_i$  using inducer  $I$ .
3. **Combine Classifiers for Final Prediction**:
  - For each new input  $x$ , gather predictions from all classifiers  $C_i(x)$ .
  - **For Classification**: Use majority voting to determine the final predicted label  $C^*(x)$ , defined as:

$$C^*(x) = \arg \max_{y \in Y} \#\{i : C_i(x) = y\}$$

where  $Y$  is the set of possible labels, and the prediction  $C^*(x)$  is the label most frequently predicted by classifiers  $C_i$ .

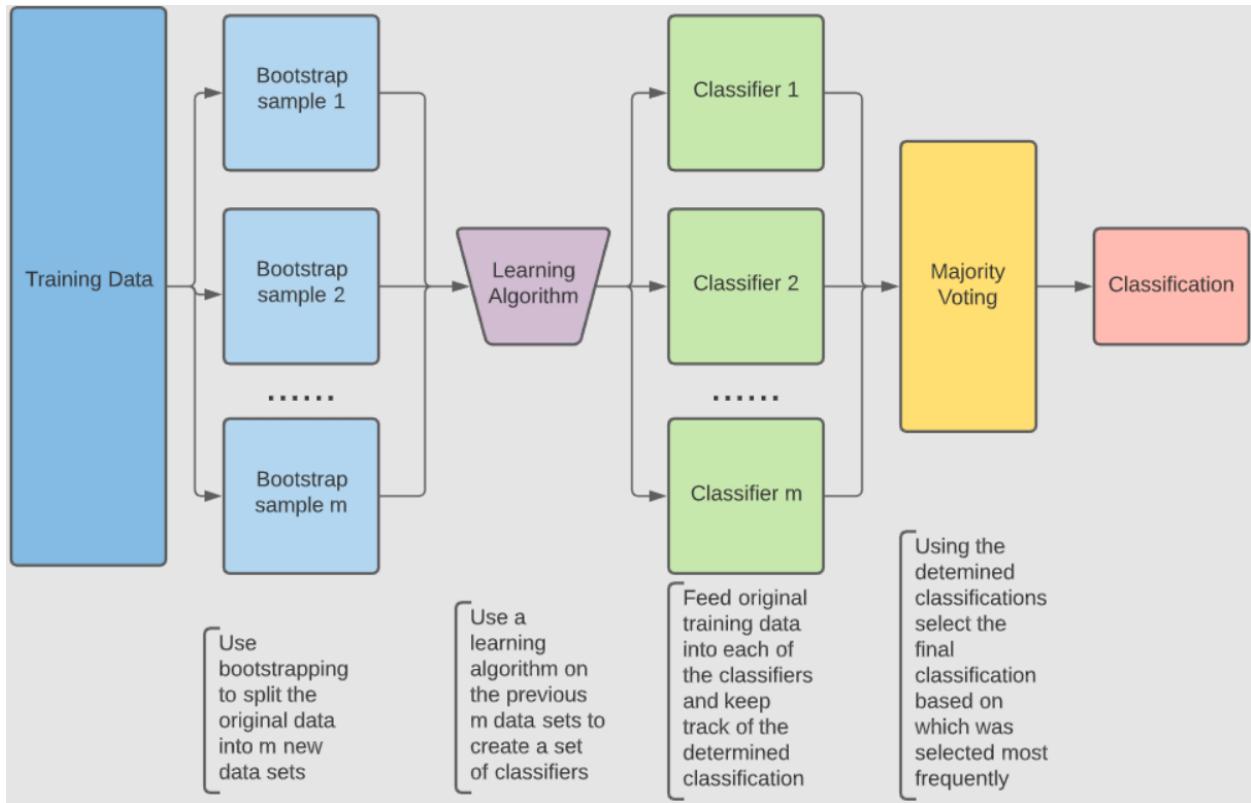


Figure 6.5: Bagging Algorithm for Classification

#### 6.4.5 Advantages and Disadvantages of Bagging

- **Advantages:**

- **Reduces Variance:** By averaging predictions over multiple models, bagging reduces the variance, which helps in achieving better generalization.
- **Minimizes Overfitting:** Bagging is effective in reducing overfitting, particularly when the base learners are prone to high variance (e.g., decision trees).
- **Parallelization:** Each model in the ensemble is trained independently on its bootstrap sample, making bagging suitable for parallel processing and thus faster on distributed systems.
- **Handles Nonlinear Relationships Well:** Bagging is highly effective in capturing complex, nonlinear patterns in the data when used with nonlinear base learners.
- **Robustness to Outliers and Noise:** The averaging effect in bagging can provide resilience to noise and outliers, as extreme predictions from individual models are smoothed out in the final prediction.
- **Improves Stability with Small Data Sets:** With bootstrap sampling, bagging can enhance the robustness of models trained on smaller data sets by creating diverse subsets, which helps mitigate issues related to limited data.

- **Disadvantages:**

- **Limited Reduction in Bias:** Bagging primarily reduces variance; it does not address the bias in base learners. If the base model has high bias, bagging will not significantly improve accuracy.
- **Reduced Interpretability:** Since bagging combines multiple models, the ensemble's interpretability is reduced. It is challenging to interpret the combined predictions in real-world applications.
- **Computationally Intensive:** Training multiple models on bootstrapped datasets requires more computational power and memory, making bagging resource-intensive, especially with a large number of base learners.
- **May Not Always Improve Performance:** If the base model is already stable with low variance, bagging may yield minimal improvements in accuracy while adding computational overhead.
- **Sensitivity to Parameter Selection:** Bagging can be sensitive to hyperparameters, such as the number of bootstrap samples and the choice of base learner, which can affect its performance and require careful tuning.
- **Potential Data Redundancy:** Bootstrap sampling with replacement may lead to data redundancy within individual models, as some instances may appear multiple times, which may not always add diversity.

## 6.5 Sequential Learning

### 6.5.1 Concept of Sequential Learning and Sequential Models

Sequential learning is essential for datasets where instances are ordered temporally or logically. Unlike traditional machine learning, which assumes data points are independently and identically distributed (i.i.d.), sequential learning models respect the order and dependencies between data points. Sequence models excel in tasks such as language modeling, speech recognition, and time-series prediction.

### 6.5.2 Sequential Data

Sequential data refers to datasets where points are dependent on neighboring points. Examples include:

- **Time-Series Data:** Predicting future stock prices based on historical trends.
- **Text:** Predicting the next word in a sentence from prior context.
- **Audio/Video Sequences:** Recognizing spoken words or identifying actions in a video.

### 6.5.3 Types of Sequential Models

Sequential models are specially adapted neural networks capable of retaining memory of prior inputs. Common models include:

#### 6.5.3.1 Recurrent Neural Networks (RNNs)

RNNs process sequences with an internal memory mechanism.

Types of RNN Architectures:

- **One-to-One:** With one input and one output, this is the classic feed-forward neural network architecture. Standard input-output mapping.
- **One-to-Many:** This is referred to as image captioning. We have one fixed-size image as input, and the output can be words or phrases of varying lengths.
- **Many-to-One:** This is used to categorize emotions. A succession of words or even subsections of words is anticipated as input. The result can be a continuous-valued regression output that represents the likelihood of having a favourable attitude.

- **Many-to-Many:** This paradigm is suitable for machine translation, such as that seen on Google Translate. The input could be a variable-length English sentence, and the output could be a variable-length English sentence in a different language. On a frame-by-frame basis, the last many to many models can be utilized for video classification.

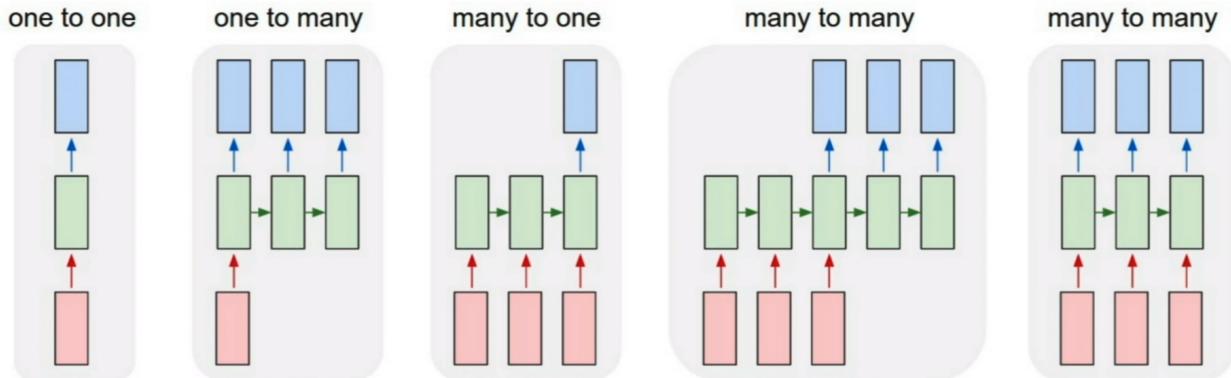


Figure 6.6: Process Sequences

Traditional RNNs, as you may know, aren't very excellent at capturing long-range dependencies. This is primarily related to the problem of vanishing gradients. Gradients or derivatives diminish exponentially as they move down the layers while training very deep networks. The problem is referred to as the ***Vanishing Gradient Problem***.

#### 6.5.3.2 Long Short-Term Memory Networks (LSTMs)

LSTMs address RNNs' long-term dependency issues with *cell states* and *gates* (input, forget, and output) that regulate information flow. They are suitable for long-sequence applications like translation.

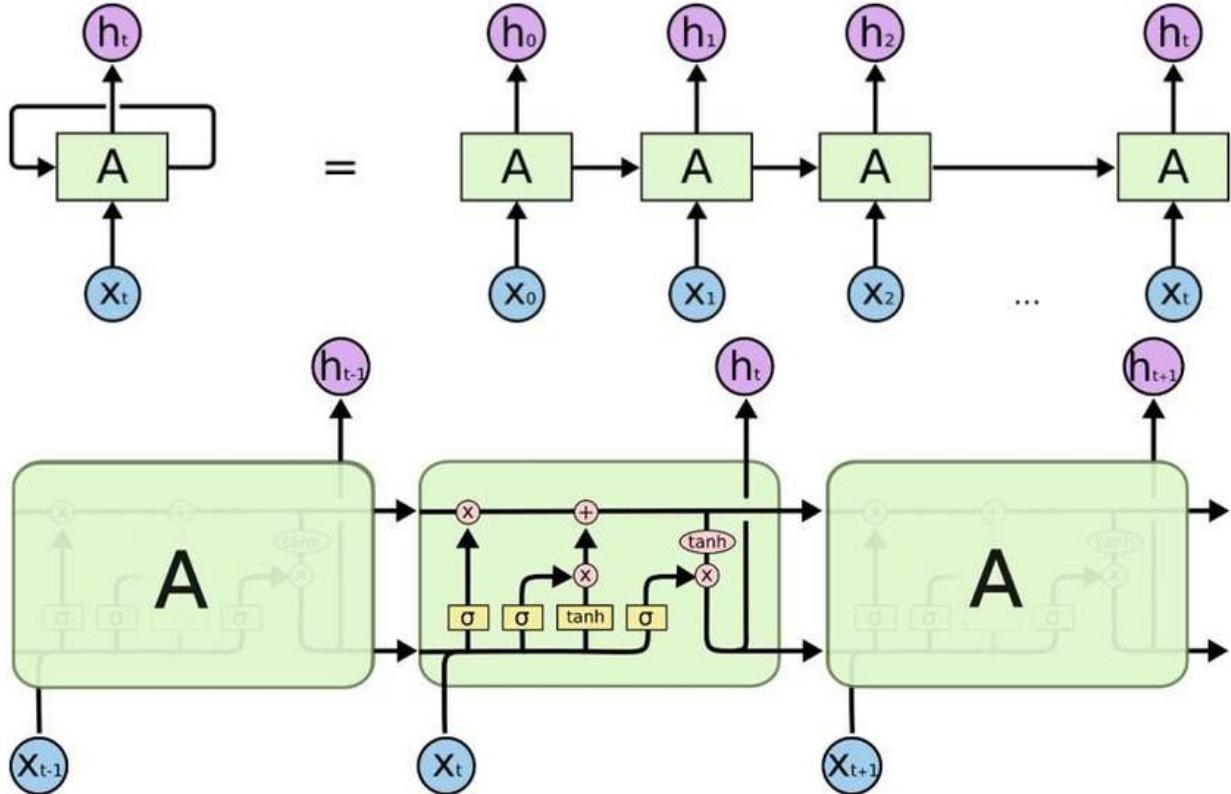


Figure 6.7: Process Sequences

### Basic Structure

At the top of the image, we observe a standard Recurrent Neural Network (RNN) structure, where each time step  $t$  is represented by a cell labeled  $A$ . Each cell processes the input  $X_t$  at that time step and passes the hidden state  $h_t$  to the next time step. This creates a sequence of computations that connect each input to subsequent ones.

RNNs pass information from one time step to the next using a single hidden state,  $h_t$ , which retains information about the previous inputs in the sequence. However, this simple structure struggles with retaining long-term dependencies due to issues like the vanishing gradient problem.

### Unfolding in Time

The RNN structure is "unfolded" across time, as shown by multiple cells (each labeled  $A$ ) connected in sequence from  $X_0$  to  $X_t$ . This representation illustrates that the network reuses the same cell architecture for each time step.

In both RNN and LSTM architectures, unfolding the network in time shows how the model processes sequences step-by-step. However, unlike basic RNNs, LSTMs have additional mechanisms to better retain long-term dependencies.

## LSTM Internal Structure

The lower part of the image focuses on the LSTM cell's internal workings. Unlike the simple RNN cell, an LSTM cell has a more complex structure with multiple gates: the forget gate, input gate, and output gate. These gates control the flow of information through the cell and allow it to retain important information for longer periods.

The LSTM cell introduces a cell state (often represented as  $C_t$ ), which runs horizontally across time steps. This cell state provides a way to retain information over long sequences, solving the vanishing gradient problem inherent in RNNs.

## Gates in LSTM

Each gate in the LSTM cell performs specific functions:

- **Forget Gate:** In the LSTM cell shown in the lower part of the image, the forget gate is represented by the symbol sigmoid ( $\sigma$ ) connected to a multiplication operation. This gate controls which parts of the cell state are discarded by scaling them down to zero or retaining them. This operation helps the cell "forget" unnecessary information from the previous time step.
- **Input Gate:** The input gate is represented by another sigmoid symbol followed by a multiplication ( $\times$ ) operation and a tanh operation. This part decides which new information will be added to the cell state. The output from the input gate (after the sigmoid and tanh transformations) is combined with the cell state, adding new relevant information.
- **Output Gate:** The output gate is shown by the third sigmoid ( $\sigma$ ) and a subsequent multiplication with the cell state after a tanh transformation. This gate determines the final hidden state  $h_t$  that will be passed on to the next time step. This hidden state is essentially the cell's output and is influenced by both the current input and the long-term information retained in the cell state.

These gates are implemented using sigmoid ( $\sigma$ ) and tanh activations, allowing the network to learn which information to retain, update, or forget as it processes each time step.

### Advantages of LSTM over RNN

- **Memory Retention:** Due to the additional cell state and gating mechanisms, LSTMs can retain important information over long time sequences, making them effective for tasks that require long-term dependencies.
- **Mitigation of Vanishing Gradients:** By carefully regulating information flow, LSTMs address the vanishing gradient issue that limits the performance of standard RNNs on long sequences.

### 6.5.3.3 Autoencoders

One of the most active study areas in Natural Language Processing is machine translation (MT) (NLP). The goal is to create a computer program that can quickly and accurately translate a text from one language (source) into another language (target) (the target). The encoder-decoder model is the fundamental architecture utilized for MT using the neural network model:

- The encoder sub-section summarizes the data in the source sentence.
- Based on the encoding, the decoder component generates the target-language output in a step-by-step manner.

The performance of the encoder-decoder network diminishes significantly as the length of the input sentence increases, which is a limitation of these approaches. The fundamental disadvantage of the earlier methods is that the encoded vector must capture the full phrase (sentence), which means that much critical information may be missed.

Furthermore, the data must “flow” through a number of RNN steps, which is challenging for large sentences. Bahdanau presented an ***attention layer*** that consists of ***attention mechanisms*** that give greater weight to some of the input words than others while translating the sentence and this gave further boost in machine translation applications.

### 6.5.3.4 Sequence-to-Sequence (Seq2Seq)

Seq2Seq models handle variable-length input and output sequences, making them essential for tasks like translation, summarization, and dialogue generation. They consist of an encoder to process the input and a decoder to generate the output.

## 6.5.4 Applications of Sequential Models

- **Natural Language Processing:** Sentiment analysis, translation, text generation.
- **Speech and Audio Processing:** Recognition and synthesis.
- **Predictive Analysis:** Stock and weather forecasting.
- **Healthcare:** Patient history analysis for diagnosis.
- **Genomics:** DNA sequence classification and prediction.

## 6.6 Boosting

### 6.6.1 History of Boosting

**Boosting** is a powerful ensemble technique in machine learning aimed at reducing both bias and variance by sequentially combining multiple weak learners to create a strong learner. The concept emerged from the question posed by Kearns and Valiant in the late 1980s: "Can a set of weak learners create a single strong learner?" Robert Schapire answered affirmatively in 1990, proving that weak learners could indeed be converted into a strong learner, laying the foundation for boosting.

Schapire and Freund later developed the *Adaptive Resampling and Combining* (Arcing) technique, which became synonymous with boosting. They introduced AdaBoost (Adaptive Boosting), one of the earliest and most impactful boosting algorithms, which iteratively focused on misclassified examples to improve accuracy. This innovation led to further developments, including Gradient Boosting and XGBoost, which are widely used today. Boosting has become essential in machine learning, offering solutions for complex tasks with high predictive accuracy.

**Overview** Boosting is an ensemble learning approach primarily used in supervised learning. It creates a sequence of models, each of which attempts to correct the errors made by the previous models. Boosting is particularly useful in scenarios where reducing bias is essential, such as classification tasks. *Unlike bagging, which averages predictions, boosting adds the outputs of the weak learners sequentially, focusing on the most challenging cases at each step.*

### 6.6.2 Sequential Learning

Boosting follows a sequential learning approach, where each model is trained to correct the errors of its predecessor. In each iteration, the algorithm assigns higher weights to misclassified examples, making them more likely to be chosen for the next model's training. This sequential dependency helps boosting focus on challenging cases, gradually refining the overall prediction accuracy.

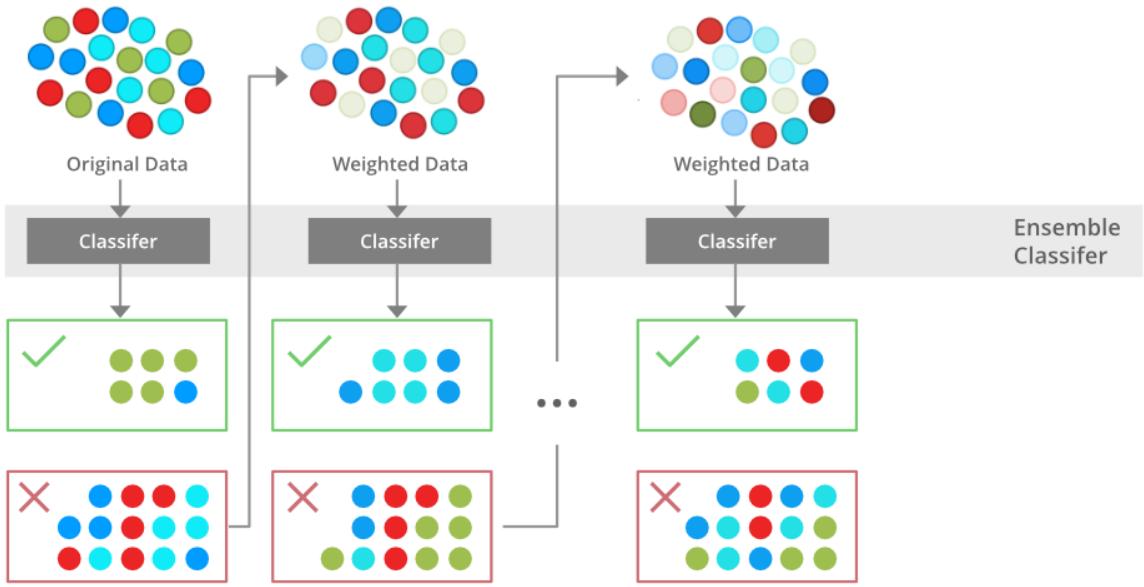


Figure 6.8: Boosting Demonstration

### 6.6.3 Weighted Training Instances in Machine Learning

In machine learning, **weighted training instances** refer to the practice of assigning a specific weight to each data point in the training set, indicating its importance during the learning process. This technique is especially prevalent in ensemble methods like boosting, where the model iteratively focuses on challenging examples to enhance overall performance.

#### Purpose of Weighted Training Instances

The primary objectives of assigning weights to training instances include:

- **Emphasizing Hard-to-Classify Instances:** Increasing the focus on examples that previous models misclassified, encouraging the algorithm to learn from its mistakes.
- **Balancing Class Distributions:** In imbalanced datasets, assigning higher weights to minority class instances to prevent the model from being biased toward the majority class.
- **Incorporating Prior Knowledge:** Reflecting the varying importance of different data points based on domain-specific insights.

#### Implementation in Boosting Algorithms

Boosting algorithms, such as AdaBoost, utilize weighted training instances to iteratively improve model accuracy:

1. **Initialization:** Assign equal weights to all training instances.
2. **Training Weak Learner:** Train a base model (weak learner) on the weighted dataset.
3. **Error Calculation:** Evaluate the model's performance, identifying misclassified instances.
4. **Weight Update:** Increase the weights of misclassified instances and decrease the weights of correctly classified ones.
5. **Iteration:** Repeat the process, with each subsequent model focusing more on previously misclassified examples.

This iterative weighting mechanism ensures that the ensemble model progressively concentrates on challenging cases, leading to improved accuracy.

### Applications and Benefits

Weighted training instances are beneficial in various scenarios:

- **Handling Imbalanced Data:** By assigning higher weights to minority class instances, models can achieve better performance on imbalanced datasets.
- **Robustness to Noise:** Reducing the influence of noisy or outlier data points by assigning them lower weights.
- **Improved Generalization:** Focusing on hard-to-classify instances helps the model generalize better to unseen data.

### Challenges of Weighted Training Instances

While weighted training instances offer significant advantages, they also present challenges:

- **Computational Complexity:** Updating weights iteratively can be computationally intensive, especially with large datasets.
- **Overfitting Risk:** Excessive focus on misclassified instances may lead to overfitting, where the model performs well on training data but poorly on new data.

#### 6.6.4 How Boosting Works

The boosting process can be outlined as follows:

1. **Initialize Weights:** Assign equal weights to all training instances.
2. **Train Weak Learner:** Develop a weak learner using the weighted dataset.
3. **Evaluate Performance:** Assess the weak learner's accuracy on the training data.

4. **Update Weights:** Increase weights for misclassified instances and decrease weights for correctly classified ones.
5. **Aggregate Learners:** Add the weak learner to the ensemble with a weight proportional to its accuracy.
6. **Iterate:** Repeat steps 2–5 until a specified number of weak learners are trained or the error reaches a minimum threshold.

This iterative approach ensures that each subsequent learner focuses more on the instances that previous learners found challenging, thereby enhancing the model's overall accuracy.

### 6.6.5 Boosting Algorithms

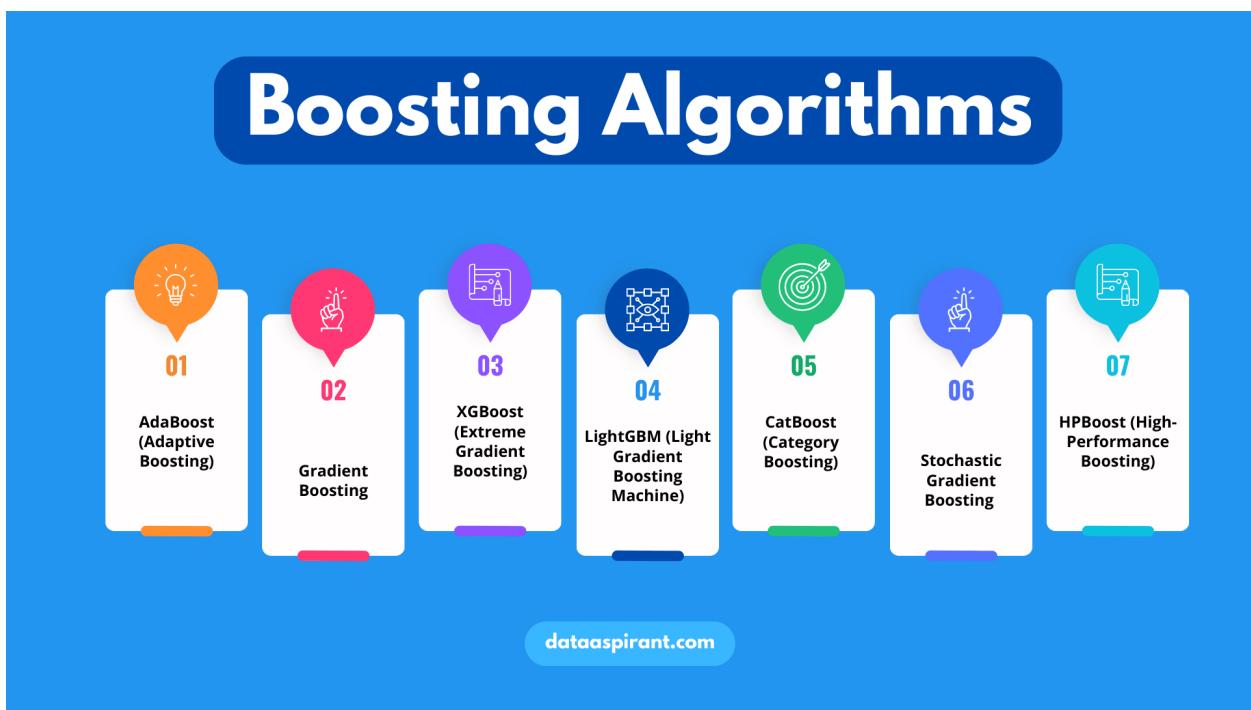


Figure 6.9: Boosting Algorithms

Several algorithms implement the boosting concept, each with unique methodologies:

#### 6.6.5.1 Adaptive Boosting (AdaBoost)

Developed by Freund and Schapire, AdaBoost combines weak classifiers sequentially, adjusting the weights of misclassified instances in each iteration. It is particularly effective for binary classification tasks and has been extended to handle multiclass and regression tasks.

## Mathematical Formulation of AdaBoost

AdaBoost adjusts the weights of training instances to focus on difficult cases:

1. **Initialization:** Assign equal weights to all instances:

$$w_i^{(1)} = \frac{1}{N} \quad \text{for } i = 1, 2, \dots, N$$

where  $N$  is the total number of training instances.

2. **Training:** Train the weak learner  $h_t$  using the weighted dataset.

3. **Error Calculation:** Compute the weighted error rate  $\epsilon_t$ :

$$\epsilon_t = \frac{\sum_{i=1}^N w_i^{(t)} \cdot I(y_i \neq h_t(x_i))}{\sum_{i=1}^N w_i^{(t)}}$$

where  $I(\cdot)$  is the indicator function,  $y_i$  is the true label, and  $h_t(x_i)$  is the prediction.

4. **Model Weight:** Determine the weight  $\alpha_t$  of the weak learner:

$$\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)$$

5. **Weight Update:** Update the weights for the next iteration:

$$w_i^{(t+1)} = w_i^{(t)} \cdot \exp(\alpha_t \cdot I(y_i \neq h_t(x_i)))$$

Normalize the weights so that they sum to 1:

$$w_i^{(t+1)} = \frac{w_i^{(t+1)}}{\sum_{j=1}^N w_j^{(t+1)}}$$

This process repeats for a predefined number of iterations or until a desired performance level is achieved.

### 6.6.5.2 Gradient Boosting

Gradient Boosting extends the boosting concept by adding models to minimize a loss function, typically through gradient descent. This approach allows for greater flexibility in error correction and is widely used for regression and classification tasks. Modern variations, such as XGBoost and LightGBM, offer optimizations that make Gradient Boosting scalable to large datasets.

## Mathematical Formulation of Gradient Boosting

Gradient Boosting builds an ensemble model by sequentially adding new models that reduce a specified loss function. The steps for Gradient Boosting are as follows:

- 1. Initialization:** Initialize the model with a constant prediction:

$$F_0(x) = \arg \min_{\gamma} \sum_{i=1}^N \mathcal{L}(y_i, \gamma)$$

where  $\mathcal{L}(y, F(x))$  is the loss function.

- 2. Compute Pseudo-Residuals:** For each instance, compute the pseudo-residuals, which represent the negative gradient of the loss function with respect to the model's predictions:

$$r_i^{(t)} = - \frac{\partial \mathcal{L}(y_i, F(x_i))}{\partial F(x_i)} \Big|_{F(x)=F_{t-1}(x)}$$

- 3. Fit a Weak Learner:** Train a weak learner  $h_t(x)$  to predict the pseudo-residuals  $r_i^{(t)}$ .
- 4. Update the Model:** Update the model by adding the predictions of the weak learner, scaled by a learning rate  $\eta$ :

$$F_t(x) = F_{t-1}(x) + \eta h_t(x)$$

where  $\eta$  is the learning rate, controlling the contribution of each weak learner.

- 5. Iterate:** Repeat steps 2–4 for a predefined number of iterations or until the loss function converges.

Gradient Boosting minimizes the loss function iteratively by fitting weak learners to the gradients, making it suitable for complex, nonlinear relationships.

### 6.6.6 Applications of Boosting

Boosting algorithms have been successfully applied in various domains:

- **Object Categorization in Computer Vision:** Boosting methods are used to combine weak classifiers based on image features, enhancing object recognition accuracy.
- **Handling Imbalanced Data:** By assigning higher weights to minority class instances, boosting helps in addressing class imbalance issues.

### 6.6.7 Challenges and Considerations

While boosting offers significant advantages, it also presents challenges:

- **Computational Complexity:** The iterative training and weight updating can be computationally intensive, especially with large datasets.

- **Overfitting Risk:** Excessive focus on misclassified instances may lead to overfitting, where the model performs well on training data but poorly on new data.

Understanding these aspects is crucial for effectively implementing boosting algorithms in machine learning tasks.

### 6.6.8 Application of Boosting in Detecting AI-Generated Text

Boosting algorithms, particularly AdaBoost and Gradient Boosting, can be highly effective for detecting AI-generated text. Since AI-generated text can often exhibit subtle, complex patterns, boosting methods allow the model to focus iteratively on the challenging cases that may be difficult to identify using simpler models. Key benefits of using boosting in this context include:

- **Enhanced Pattern Recognition:** By iteratively focusing on misclassified examples, boosting helps the model capture nuanced patterns typical of AI-generated text.
- **Reduced Bias:** Boosting helps reduce bias, which is beneficial when the AI-generated and human-generated text may be quite similar. The model can iteratively adjust to capture subtle discrepancies.
- **Customizable Loss Function:** With Gradient Boosting, we can customize the loss function to focus specifically on characteristics that differentiate AI-generated text, such as unusual phrase patterns or lexical diversity.
- **High Predictive Accuracy:** Boosting algorithms, especially XGBoost and CatBoost, have a strong track record for classification tasks in NLP, making them suitable for AI-generated text detection.

#### *Example*

Suppose we have a dataset containing both AI-generated and human-generated text. By training a boosting model (e.g., AdaBoost) with decision tree stumps as weak learners, the model could learn to focus on common misclassification cases, such as syntactically correct but semantically odd phrases, which are more common in AI-generated text.

**Summary** Boosting offers a robust approach for distinguishing between AI-generated and human-generated text by focusing on hard-to-detect patterns, making it a valuable tool for maintaining content authenticity and improving AI accountability in natural language processing tasks.

## 6.7 Gradient Boosting

### 6.7.1 Overview and History

Gradient Boosting is a powerful machine learning technique that constructs predictive models by sequentially adding weak learners, typically decision trees, to minimize a specified loss function. This approach improves model accuracy by correcting the errors made by previous learners through an iterative process. Gradient Boosting was introduced by Jerome Friedman in 1999 and has become foundational for regression, classification, and ranking tasks.

The concept of Gradient Boosting originated from the observation by Leo Breiman that boosting can be interpreted as an optimization algorithm on a cost function. This led to the development of explicit regression Gradient Boosting algorithms by Friedman, along with the more general functional gradient boosting perspective introduced by Mason, Baxter, Bartlett, and Frean. Their work viewed boosting algorithms as iterative functional gradient descent algorithms in function space. Gradient Boosting generalizes the concept by optimizing arbitrary differentiable loss functions, allowing for broad applications beyond regression, including classification and ranking problems.

### 6.7.2 Loss Function

The loss function in Gradient Boosting defines the error or residual to be minimized at each iteration. Different loss functions can be chosen for various tasks, influencing the model's sensitivity to outliers and overall accuracy.

#### 6.7.2.1 Regression

In regression tasks, common loss functions include:

- **Mean Squared Error (MSE):** This loss function is the average squared difference between the predicted and actual values. It is widely used due to its sensitivity to large errors, making it effective for regression problems where large deviations are penalized.

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- **Mean Absolute Error (MAE):** MAE measures the average absolute difference between the predicted and actual values. It is less sensitive to large outliers, making it suitable for tasks where robustness against extreme deviations is necessary.

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

- **Huber Loss:** Huber loss combines MSE and MAE and is less sensitive to outliers than MSE while retaining some sensitivity to large deviations. It transitions from MSE to MAE depending on the residual size and is defined as:

$$\mathcal{L}(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

The choice of loss function can be based on the specific problem and its robustness to outliers.

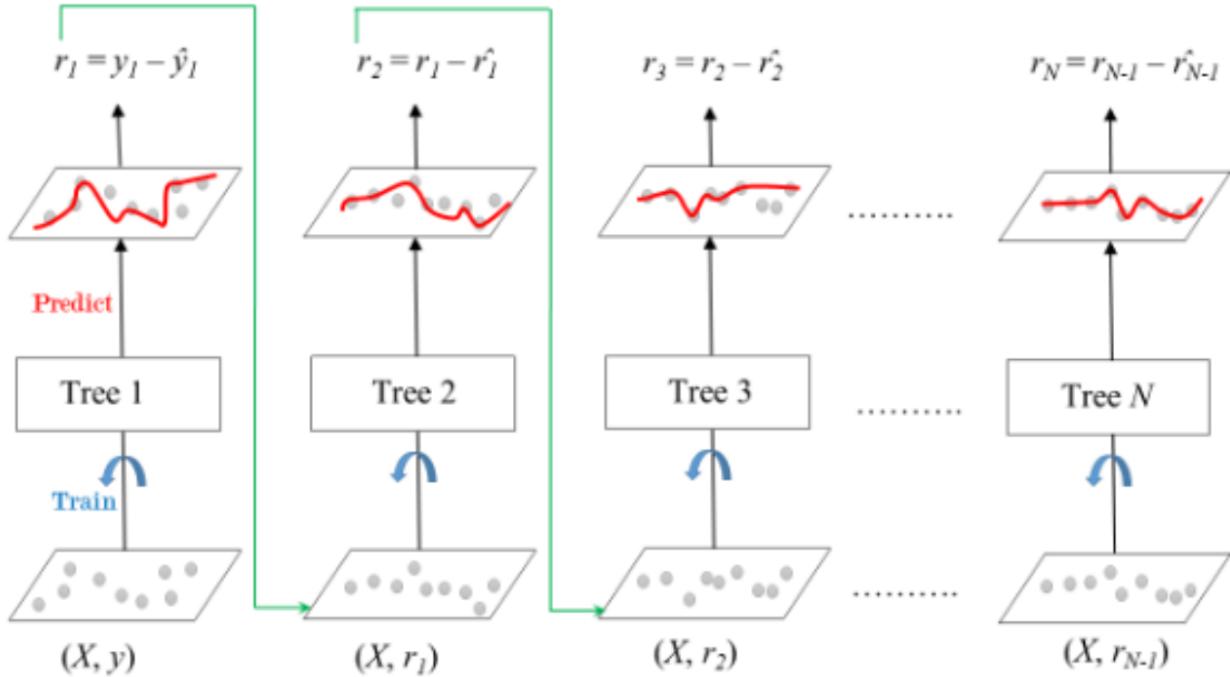


Figure 6.10: Architecture of Gradient Boosting

**Description and Comments** This figure provides a step-by-step illustration of the Gradient Boosting algorithm, where each tree is trained sequentially on the residuals (errors) of the previous tree, ultimately improving the model's prediction accuracy. The process is depicted as follows:

- **Initial Step:** The first decision tree, denoted as *Tree 1*, is trained on the original data points  $(X, y)$ , where  $X$  is the set of features and  $y$  is the target variable. The model makes initial predictions  $\hat{y}_1$ , which have associated residuals  $r_1 = y - \hat{y}_1$ .
- **Subsequent Trees (Residual Learning):** For each subsequent tree, the algorithm fits the model to the residuals of the previous prediction. Tree 2 is trained on the residuals from Tree 1, denoted as  $r_1$ . Similarly, Tree 3 is trained on the residuals  $r_2$  left by Tree 2, and this pattern continues through the  $N$ -th tree.

Mathematically, the residuals  $r_t$  for each tree  $t$  are computed as:

$$r_t = r_{t-1} - \hat{r}_{t-1},$$

where  $\hat{r}_{t-1}$  represents the prediction of the residuals by the  $(t - 1)$ -th tree.

- **Error Reduction (Gradient Direction):** The goal of each tree in this sequence is to minimize the prediction error by reducing the residuals. Each tree makes adjustments in the gradient direction to correct the errors made by the previous trees, effectively "boosting" the accuracy of the overall ensemble model.
- **Final Prediction (Ensemble):** After  $N$  trees have been trained, the final prediction is obtained by combining the predictions of all individual trees. The ensemble model aggregates these predictions, typically through a weighted sum, to make the final prediction, which is a refined and accurate approximation of the target variable  $y$ .

### 6.7.2.2 Classification

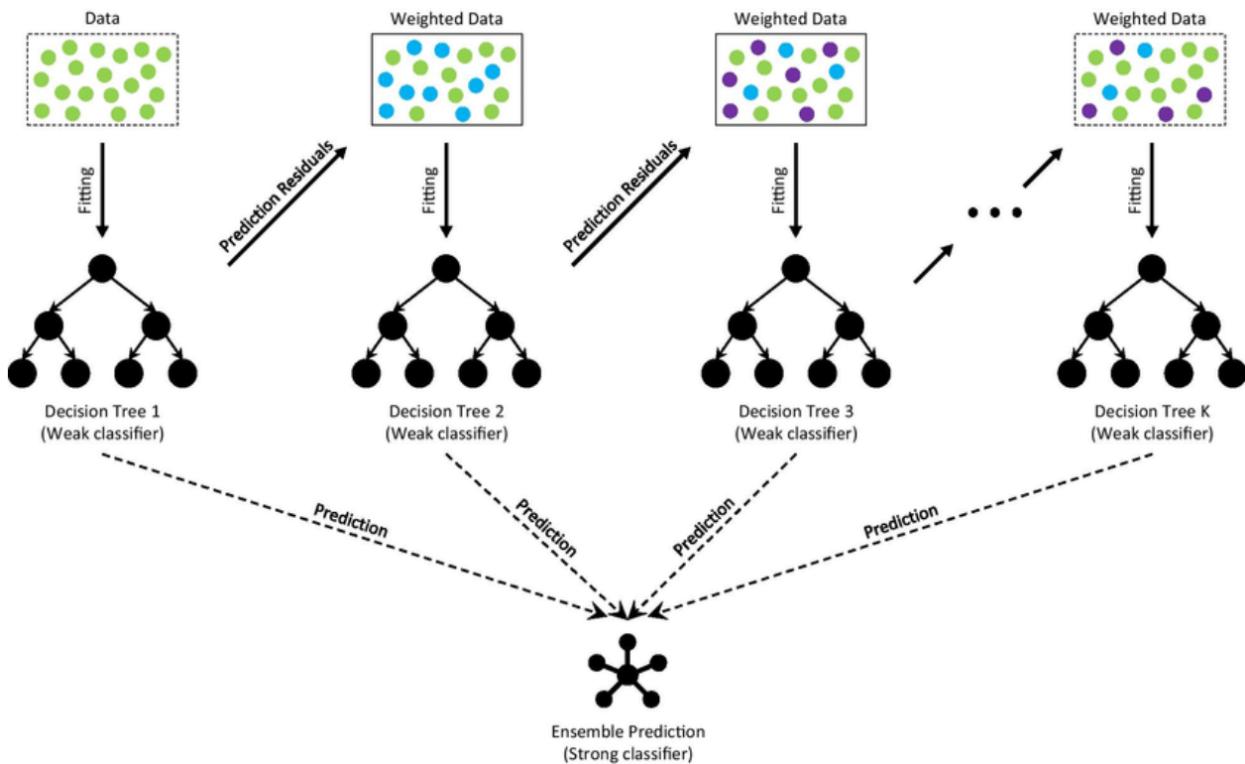


Figure 6.11: Architecture of Gradient Boosting

**Description:** The figure illustrates the process of building an ensemble model using Gradient Boosting with Decision Trees as weak classifiers. Each decision tree in the sequence is trained to fit the residuals (errors) of the previous trees, resulting in an incremental improvement in the model's predictive accuracy.

- **Data and Residuals:** Initially, the model is trained on the full dataset. After each iteration, residuals (prediction errors) from the previous tree are calculated and used to weight the data for the subsequent tree.
- **Weak Classifiers (Decision Trees):** Each step in the boosting process adds a new weak classifier, typically a decision tree. The trees are denoted as Decision Tree 1, Decision Tree 2, ..., Decision Tree K. Each of these trees focuses on learning from the residuals of the previous ensemble.
- **Weighted Data:** With each iteration, the data points are reweighted based on the residuals from the previous tree. Points with higher residuals are given more weight so that subsequent trees focus more on these challenging instances.
- **Ensemble Prediction:** Once all the weak classifiers have been trained, their predictions are combined to form a strong classifier. The final prediction of the ensemble is a weighted sum of the predictions from each individual tree, resulting in a more accurate overall model.

This process continues iteratively, with each new tree aiming to correct the mistakes of the preceding trees, thereby reducing the overall prediction error and improving the model's accuracy.

In classification tasks, common loss functions include:

- **Logistic Loss (Log Loss):** For binary classification, logistic loss measures performance by penalizing predictions that are probabilistically far from actual class labels. It is especially common in binary classification tasks.

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

- **Multinomial Deviance:** Extending logistic loss to multiclass classification, multinomial deviance calculates error across multiple classes by penalizing probabilistic deviations.

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K y_{i,k} \log(\hat{y}_{i,k})$$

where  $K$  is the number of classes, and  $y_{i,k}$  is a binary indicator (0 or 1) indicating if class label  $k$  is the correct classification for observation  $i$ .

Selecting the right loss function is key to maximizing model accuracy and effectiveness.

### 6.7.3 Additive Model

Gradient Boosting constructs an additive model by sequentially adding weak learners to minimize the loss function. Each learner is trained on the residuals of previous learners,

improving the overall model iteratively.

Mathematically, the additive model can be described as:

$$F_M(x) = F_0(x) + \sum_{m=1}^M \eta h_m(x)$$

where:

- $F_M(x)$  is the final model after  $M$  iterations.
- $F_0(x)$  is the initial model, typically a constant.
- $\eta$  is the learning rate, determining each weak learner's contribution.
- $h_m(x)$  is the  $m$ -th weak learner.

The learning rate  $\eta$  is crucial, balancing between the number of iterations and the contribution of each learner. A smaller  $\eta$  often requires more iterations but can lead to better generalization by reducing overfitting.

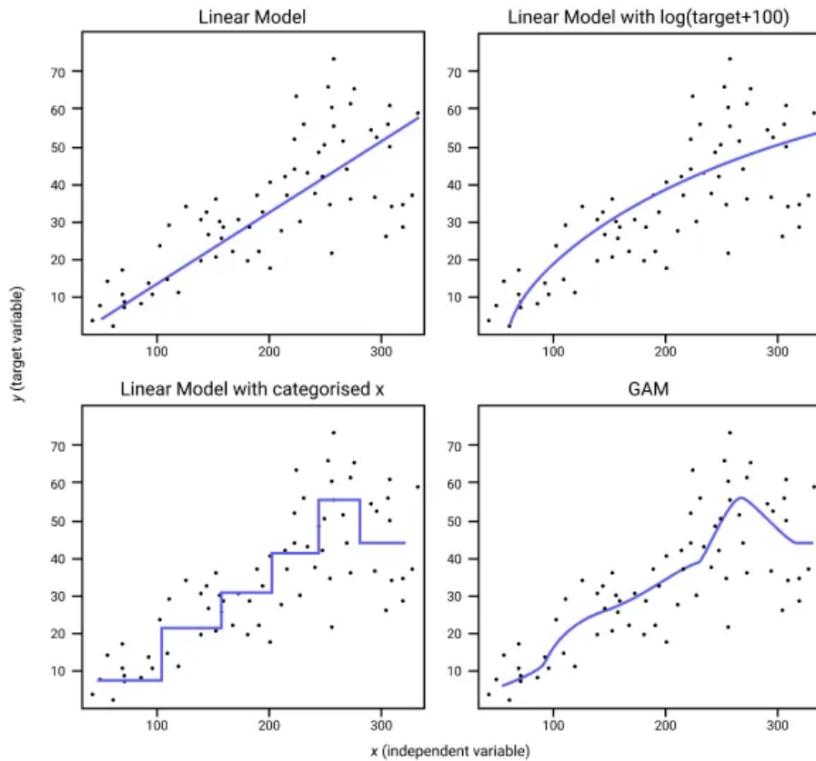


Figure 6.12: Generalized Additive Models

**Description and Explanation:** This figure compares the fits of different modeling approaches on a dataset with an independent variable  $x$  and a target variable  $y$ . The models compared are as follows:

- **Linear Model (Top Left):** This model represents a simple linear regression fit. The straight line indicates that this model assumes a constant linear relationship between  $x$  and  $y$ . While it captures the general upward trend, it fails to capture any non-linear patterns.
- **Linear Model with Log Transformation of Target (Top Right):** Here, a linear model is applied after a log transformation on the target variable,  $\log(y + 100)$ . The transformation allows the model to capture some of the non-linear pattern present in the data, resulting in a curve rather than a straight line. This approach helps when the relationship between  $x$  and  $y$  is non-linear, but it does not capture all details of the data distribution.
- **Linear Model with Categorized  $x$  (Bottom Left):** In this approach, the continuous variable  $x$  is categorized into discrete bins, and a linear model is fit on these categories. The result is a stepwise function, where each step represents the average response for that category. This approach can capture sudden changes and plateaus in the data but may miss finer details.
- **Generalized Additive Model (GAM) (Bottom Right):** The Generalized Additive Model (GAM) allows for a more flexible, non-linear relationship between  $x$  and  $y$  by fitting smooth functions to subsets of the data. The blue curve represents the model's ability to adapt to the data's underlying pattern more accurately than the previous approaches. GAM is useful for capturing complex, non-linear relationships without making strong assumptions about the form of the relationship.

#### 6.7.4 Gradient Descent in Gradient Boosting

Gradient Boosting minimizes the loss function using gradient descent by iteratively adding weak learners that move in the direction of the negative gradient of the loss function. This process enables the model to adaptively refine predictions by focusing on residual errors.

The process can be summarized as follows:

1. **Initialize** the model with a constant prediction (such as the mean for regression or log-odds for classification).
2. **Compute the negative gradient** of the loss function with respect to the current model's predictions to identify the residuals.
3. **Fit a weak learner** to the residuals, effectively addressing the errors from the prior model.

4. **Update the model** by scaling and adding the weak learner with a learning rate  $\eta$ :

$$F_m(x) = F_{m-1}(x) + \eta h_m(x)$$

where  $F_m(x)$  is the updated model,  $F_{m-1}(x)$  is the previous model, and  $h_m(x)$  is the weak learner fitted to the residuals.

5. **Repeat** until convergence or a specified iteration count.

**More details:**

- **Input:** Training set  $\{(x_i, y_i)\}_{i=1}^n$ , a differentiable loss function  $\mathcal{L}(y, F(x))$ , number of iterations  $M$ .
- **Algorithm:**

1. Initialize the model with a constant value:

$$F_0(x) = \arg \min_{\gamma} \sum_{i=1}^n \mathcal{L}(y_i, \gamma).$$

2. For  $m = 1$  to  $M$ :

(a) Compute the pseudo-residuals:

$$r_{im} = - \left[ \frac{\partial \mathcal{L}(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \dots, n.$$

(b) Fit a weak learner  $h_m(x)$  to the pseudo-residuals, using the training set  $\{(x_i, r_{im})\}_{i=1}^n$ .

(c) Compute the multiplier  $\gamma_m$  by solving:

$$\gamma_m = \arg \min_{\gamma} \sum_{i=1}^n \mathcal{L}(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)).$$

(d) Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output  $F_M(x)$ .

### 6.7.5 Regularization Techniques

Regularization in Gradient Boosting is essential to prevent overfitting, improving the generalizability of the model on unseen data. Various techniques include:

#### 6.7.5.1 Shrinkage

Shrinkage, or learning rate regularization, modifies the update rule as:

$$F_m(x) = F_{m-1}(x) + \nu \cdot \gamma_m h_m(x), \quad 0 < \nu \leq 1$$

where  $\nu$  is the learning rate. Small learning rates (e.g.,  $\nu < 0.1$ ) yield better generalization but require more iterations.

#### 6.7.5.2 Stochastic Gradient Boosting

Stochastic Gradient Boosting is a variant of the standard Gradient Boosting algorithm introduced by Friedman. This modification is inspired by Breiman's bootstrap aggregation, or "bagging," method. *Specifically, it proposes that at each iteration, a base learner is fit on a random subsample of the training data without replacement.* Friedman observed a substantial *improvement in Gradient Boosting's accuracy with this modification.*

The **subsample size** is represented by a **constant fraction  $f$  of the total training set size**. When  $f = 1$ , the algorithm operates deterministically and is identical to the standard Gradient Boosting algorithm. However, smaller values of  $f$  introduce randomness, which helps prevent overfitting and acts as a regularization technique. Additionally, the algorithm becomes faster, as each iteration uses a smaller dataset to fit the regression trees. Typically, **a subsample fraction  $0.5 \leq f \leq 0.8$  provides good results for small to moderate-sized training sets.** Setting  $f = 0.5$  means that half of the training data is used to build each base learner.

Like bagging, subsampling in Stochastic Gradient Boosting allows for the definition of an *out-of-bag error*, which estimates the prediction improvement by evaluating the model on observations not used in the training of the base learner. This approach can replace an independent validation dataset, though out-of-bag estimates may underestimate actual performance improvements and the optimal number of iterations.

#### 6.7.5.3 Number of Observations in Leaves

Gradient Tree Boosting implementations often include *regularization by limiting the minimum number of observations in the terminal nodes (leaves) of trees*. This constraint prevents splits that would result in nodes containing fewer than the specified minimum number of samples, thereby helping to reduce variance in predictions at leaves. By setting this limit, the model can achieve more stable and generalizable predictions.

#### 6.7.5.4 Complexity Penalty

Another regularization technique in Gradient Boosting is to penalize model complexity. For Gradient Boosted Trees, ***model complexity can be defined in terms of the proportional number of leaves in the trees.*** A complexity penalty optimizes both the loss and the model's structural complexity, which corresponds to a post-pruning approach where branches that fail to reduce the loss by a threshold are removed.

Additional regularization techniques, such as applying an  $\ell_2$  penalty on the leaf values, can also be used to prevent overfitting. This penalization helps in achieving a balance between model flexibility and generalization ability.

Each method contributes to the model's robustness, helping balance between fitting the data closely and generalizing well.

#### 6.7.6 Feature Importance and Interpretability

Gradient Boosting can provide feature importance rankings, which help interpret which variables contribute most to the model's predictions. This is typically done by aggregating the importance metrics of base learners, making it useful for understanding the relative impact of features in complex datasets. Despite these insights, Gradient Boosting's ensemble nature can limit interpretability due to the complexity of the combined models.

#### 6.7.7 Comparison between AdaBoost and Gradient Boosting

AdaBoost	Gradient Boosting
During each iteration in AdaBoost, the weights of incorrectly classified samples are increased, so that the next weak learner focuses more on these samples.	Gradient Boosting updates the weights by computing the negative gradient of the loss function with respect to the predicted output.
AdaBoost uses simple decision trees with one split known as the decision stumps of weak learners.	Gradient Boosting can use a wide range of base learners, such as decision trees and linear models.
AdaBoost is more susceptible to noise and outliers in the data, as it assigns high weights to misclassified samples.	Gradient Boosting is generally more robust, as it updates the weights based on the gradients, which are less sensitive to outliers.

Table 6.1: Difference between AdaBoost and Gradient Boosting

#### 6.7.8 Variants of Gradient Boosting

Over time, several variants of Gradient Boosting have emerged, optimizing the standard algorithm for specific applications and computational efficiency:

#### 6.7.8.1 LightGBM

LightGBM (Light Gradient Boosting Machine) is a high-performance gradient boosting framework developed by Microsoft, optimized for speed and efficiency on large-scale datasets. Unlike traditional gradient boosting algorithms, LightGBM introduces several key innovations:

- **Leaf-Wise Tree Growth:** LightGBM grows trees leaf-wise rather than level-wise, splitting the leaf with the largest loss reduction, often resulting in higher accuracy and deeper trees.
- **Histogram-Based Algorithm:** Continuous feature values are bucketed into discrete bins, significantly reducing memory usage and computation time.
- **Gradient-Based One-Side Sampling (GOSS):** LightGBM retains only the data points with large gradients, focusing on the most informative samples to speed up training without sacrificing performance.
- **Exclusive Feature Bundling (EFB):** LightGBM combines sparse features into a single feature, reducing the number of features and improving efficiency.

LightGBM is known for its speed and low memory usage, making it suitable for large, high-dimensional datasets. Its ability to handle sparse data, support parallel and distributed learning, and leverage modern hardware like GPUs has made it popular in applications such as financial modeling, ranking problems, and real-time prediction systems.

#### 6.7.8.2 CatBoost

CatBoost is an open-source gradient boosting library developed by Yandex, initially released in July 2017. Unlike traditional gradient boosting frameworks, CatBoost is specifically designed to handle categorical features efficiently using a permutation-driven approach, which helps mitigate overfitting issues. Key features of CatBoost include:

- **Native Handling of Categorical Features:** CatBoost natively supports categorical features, making it a powerful tool for datasets with non-numeric data.
- **Ordered Boosting:** This technique helps to reduce overfitting by using ordered statistics to process categorical data.
- **Oblivious Trees:** CatBoost utilizes symmetric, or oblivious, trees for faster execution.
- **Fast GPU Training:** Optimized for GPU training, CatBoost provides efficient training on large datasets.
- **Cross-Platform and Language Support:** CatBoost supports Python, R, C++, Java, and models can be exported to other formats such as ONNX and Core ML.

CatBoost has gained recognition and is widely used in the machine learning community, ranking among the top frameworks in Kaggle's surveys and receiving awards like InfoWorld's "Best Machine Learning Tools" in 2017.

#### 6.7.8.3 XGBoost

XGBoost (eXtreme Gradient Boosting) is an open-source gradient boosting library initially released in March 2014. Developed by Tianqi Chen and maintained by the Distributed (Deep) Machine Learning Community (DMLC), XGBoost is known for its efficiency, scalability, and competitive performance in machine learning competitions. XGBoost supports a wide range of languages, including C++, Python, R, Java, and Scala, and can operate on Linux, macOS, and Windows. It is compatible with single-machine setups as well as distributed frameworks like Apache Hadoop, Apache Spark, Apache Flink, and Dask.

Some unique features of XGBoost include:

- **Regularization:** XGBoost applies clever penalization to tree structures to prevent overfitting.
- **Newton Boosting:** A second-order gradient boosting technique for more accurate optimization.
- **Sparsity-Aware Algorithms:** Efficient handling of sparse data with parallel tree boosting.
- **Leaf Node Shrinking:** Proportionally shrinks leaf nodes to improve model generalization.
- **Integration with Distributed Systems:** Supports scalable, distributed training across various data processing frameworks.

XGBoost gained popularity due to its high accuracy and has been the preferred algorithm for many winning solutions in ML competitions, despite the reduced interpretability compared to simpler models like single decision trees.

#### 6.7.9 Applications of Gradient Boosting

Gradient Boosting has widespread applications in both industry and academia. Some of its common use cases include:

- **Financial Modeling:** Used in credit scoring, risk assessment, and fraud detection, where accuracy is paramount.
- **Healthcare:** Applied in predictive diagnostics, personalized treatment recommendations, and analyzing complex medical datasets.

- **E-commerce:** Employed in recommendation systems, demand forecasting, and pricing optimization to enhance customer experience.
- **Natural Language Processing (NLP):** Utilized in text classification, sentiment analysis, and language translation.
- **Ranking and Search Engines:** Many search engines, such as Yahoo and Yandex, utilize Gradient Boosting in their ranking algorithms for more accurate search results.
- **Scientific Research:** High Energy Physics experiments, like those at the Large Hadron Collider, use Gradient Boosting to analyze particle collisions and confirm theoretical predictions.

Gradient Boosting's flexibility and power make it a preferred method for various predictive modeling applications.

### 6.7.10 Disadvantages of Gradient Boosting

While Gradient Boosting can significantly improve the accuracy of weak learners, it also has notable drawbacks:

- **Computational Intensity:** Gradient Boosting can be slow to train, especially with a large number of iterations or complex base learners.
- **Overfitting Risk:** Without careful regularization, it can overfit the training data, especially if the model complexity is high.
- **Interpretability:** As an ensemble of weak learners, Gradient Boosting sacrifices interpretability. Following the decision path across hundreds or thousands of trees is difficult.

Some advanced techniques, such as model compression or surrogate models, have been developed to approximate the decision function of Gradient Boosting with simpler, interpretable models.

## 6.8 Extreme Gradient Boosting (XGBoost)

### 6.8.1 Classification and Regression Trees (CART)

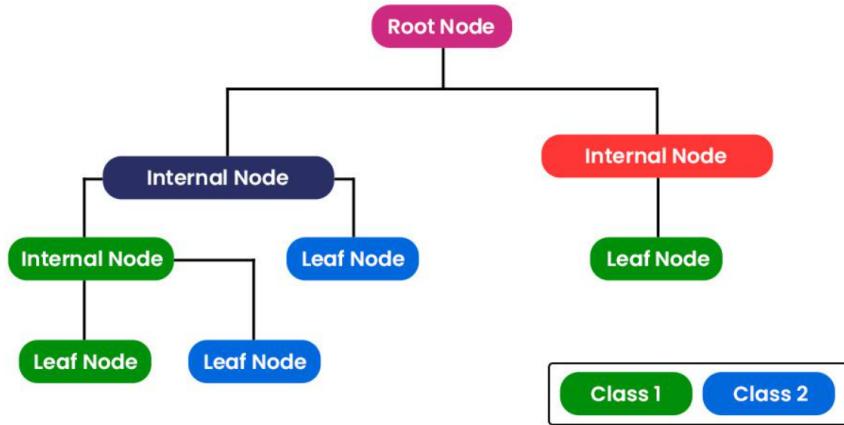


Figure 6.13: CART

XGBoost uses Classification and Regression Trees (CART) as base learners to build its model. Unlike traditional decision trees, which focus solely on boundaries for classification tasks, CART in XGBoost optimizes by minimizing a specified loss function, making it versatile for both classification and regression tasks. The CART algorithm, developed by Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone in 1984, is a foundational machine learning model used for predictive tasks.

#### 6.8.1.1 Overview of CART in XGBoost

CART is a form of decision tree algorithm that builds a tree structure to predict a target variable by recursively splitting data at nodes based on an optimal split. Each split is determined using criteria that maximize homogeneity within the resulting subsets. CART can handle both categorical target variables (Classification Trees) and continuous target variables (Regression Trees).

#### 6.8.1.2 Structure and Function of CART

A CART model is structured as a binary tree with nodes representing decision points and branches corresponding to the possible outcomes of each decision. The key features include:

- **Root Node:** Represents the entire dataset, serving as the initial decision point.
- **Internal Nodes:** Represent split points based on predictor variables, dividing the data into subgroups.

- **Leaf Nodes:** Terminal nodes that contain the predicted outcome, either a class label (for classification) or a continuous value (for regression).

#### 6.8.1.3 Types of Trees in CART

CART consists of two main types of trees:

- **Classification Trees:** Designed for categorical target variables, where the algorithm assigns a class label to each leaf node. These trees use criteria like Gini impurity to determine the best split, ensuring the purity of each subset.
- **Regression Trees:** Built for continuous target variables, where each leaf node represents a predicted value. The trees are split using criteria based on residuals, aiming to minimize the difference between the actual and predicted values.

#### 6.8.1.4 Splitting Criteria in CART

CART uses a greedy algorithm to determine the optimal split at each node by maximizing homogeneity in the resulting subgroups. The splitting criteria differ based on the type of task:

- **Classification (Gini Impurity):** The Gini impurity measures the probability of misclassifying a randomly chosen element from a subset. It is computed as:

$$\text{Gini} = 1 - \sum_{i=1}^n p_i^2$$

where  $p_i$  represents the probability of an instance being classified to a particular class. A lower Gini impurity indicates higher purity within the subset.

- **Regression (Residual Reduction):** For regression tasks, CART uses residual reduction, which aims to minimize the sum of squared errors within each subset. The goal is to achieve splits that result in subsets with minimal variance.

#### 6.8.1.5 Algorithm and Steps for CART

The CART algorithm proceeds as follows:

1. **Identify Best Split Point for Each Input:** Each feature is evaluated to find the best split point that maximizes homogeneity in the subsets.
2. **Select the Optimal Split:** Based on the candidate splits, the algorithm selects the one that best reduces impurity or residuals, depending on the task.
3. **Recursive Splitting:** The chosen split divides the data, and the process is repeated for each subset until stopping criteria are met.

4. **Stopping Criteria:** The process halts when further splits do not significantly reduce impurity, or when the maximum tree depth is reached.

#### 6.8.1.6 Pruning in CART

To avoid overfitting, CART often uses pruning techniques to simplify the tree. Two common approaches are:

- **Cost Complexity Pruning:** This involves calculating the cost of each node and removing nodes that provide minimal gain.
- **Information Gain Pruning:** Nodes are pruned based on the information gain they provide, with nodes having low information gain being removed.

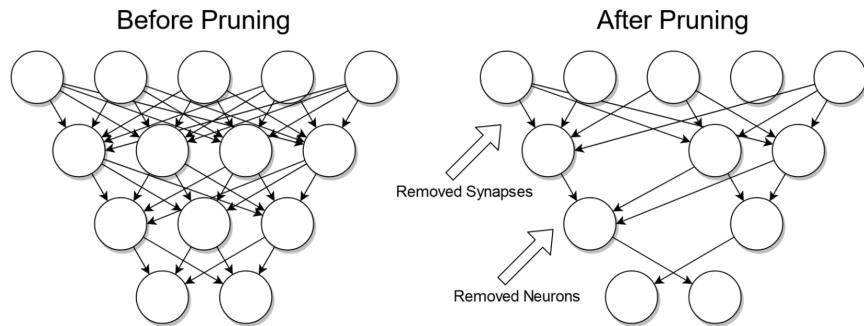


Figure 6.14: Pruning in CART

**Description:** This figure illustrates the concept of pruning in neural networks, which is a technique used to reduce the complexity of a network by removing unnecessary elements. The left side shows a fully connected network **before pruning**, while the right side shows the same network **after pruning**.

- **Before Pruning:** The network is fully connected, meaning each neuron in one layer is connected to every neuron in the next layer. This setup often leads to high computational and memory requirements due to the large number of connections (synapses) and neurons.
- **After Pruning:** Pruning removes certain neurons and synapses that are determined to be non-essential to the network's performance. Removed synapses are depicted with arrows pointing to the empty connections, while removed neurons are shown with arrows pointing to the empty circles where nodes previously existed.

### 6.8.1.7 Gini Impurity in CART

The Gini impurity index, used in classification tasks, is a measure of the probability of misclassifying a randomly chosen element. Its formula is given by:

$$\text{Gini} = 1 - \sum_{i=1}^n (p_i)^2$$

where  $p_i$  is the probability of an object being classified to a specific class. The closer the Gini index is to zero, the purer the subset.

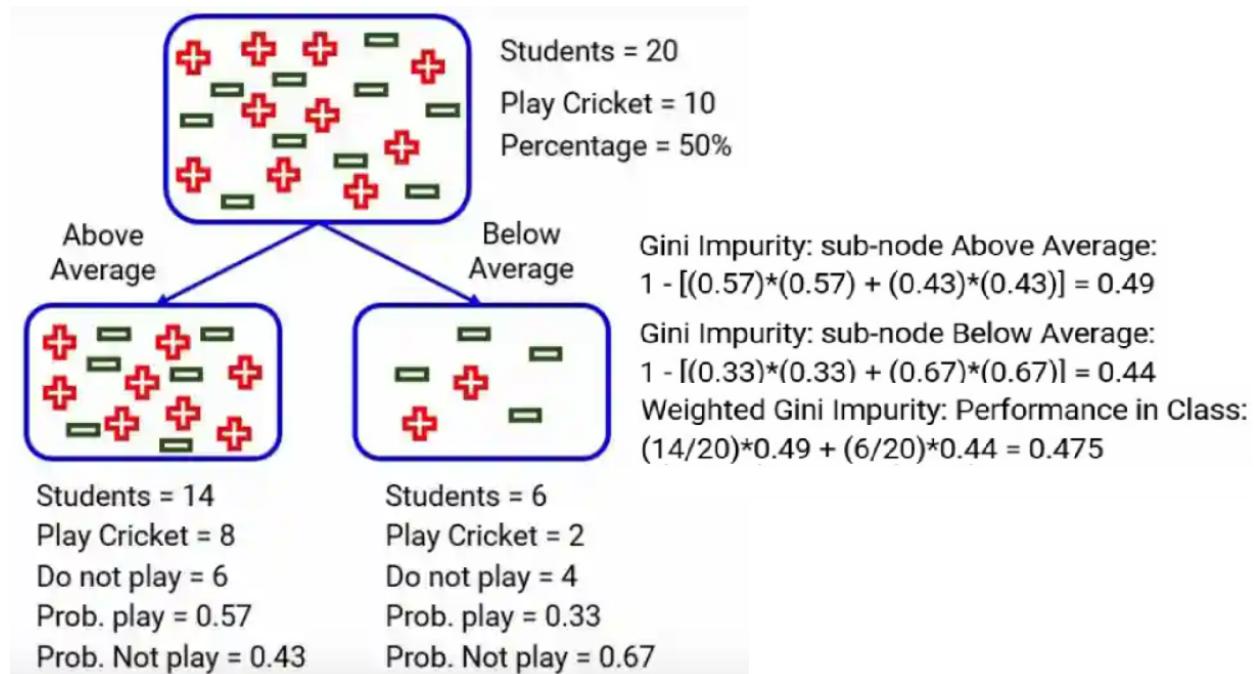


Figure 6.15: Gini Impurity Calculation

### 6.8.1.8 Exact Greedy Algorithm

The Exact Greedy Algorithm in XGBoost exhaustively searches for the best possible split at each node of the tree by evaluating all potential split points across all features. It selects the split that maximizes the gain in the objective function, which is critical for optimal partitioning of the data.

The gain from a split can be defined mathematically as follows:

$$\text{Gain} = \frac{1}{2} \left( \frac{(\sum_{i \in L} g_i)^2}{\sum_{i \in L} h_i + \lambda} + \frac{(\sum_{i \in R} g_i)^2}{\sum_{i \in R} h_i + \lambda} - \frac{(\sum_i g_i)^2}{\sum_i h_i + \lambda} \right)$$

where:

- $L$  and  $R$  represent the left and right child nodes after a potential split.

- $g_i$  is the gradient of the loss function with respect to the prediction for instance  $i$ .
- $h_i$  is the Hessian, or the second derivative of the loss function.
- $\lambda$  is a regularization term that helps prevent overfitting.

The use of both the gradient and Hessian allows for a more nuanced understanding of the loss landscape, enabling better decision-making for splits. However, this algorithm can be computationally intensive because it requires sorting the feature values and evaluating every possible split for all features.

#### 6.8.1.9 Approximate Algorithm

To mitigate the computational burden of the exact greedy approach, XGBoost implements an Approximate Algorithm. This algorithm generates candidate splits based on the distribution of feature values and uses a method of aggregating gradient statistics to find the best split from these candidates.

The approximation process can be summarized as follows:

- 1. Propose Candidate Splits:** The algorithm divides the feature values into buckets based on percentiles (e.g., 25th, 50th, 75th percentiles). This step reduces the number of possible splits significantly while still capturing the overall distribution of the data.
- 2. Aggregate Gradient Statistics:** For each bucket, it calculates the accumulated gradient ( $G_k$ ) and Hessian ( $H_k$ ) statistics:

$$G_k = \sum_{j \in \{j | s_k, v \geq x_{jk} > s_k, v-1\}} g_j$$

$$H_k = \sum_{j \in \{j | s_k, v \geq x_{jk} > s_k, v-1\}} h_j$$

- 3. Select the Best Split:** Using the aggregated statistics, the algorithm evaluates splits based on the gain formula, akin to the exact greedy method, but only considers the candidates derived from the buckets:

$$\text{Gain}_{\text{approx}} = \frac{1}{2} \left( \frac{(G_L)^2}{H_L + \lambda} + \frac{(G_R)^2}{H_R + \lambda} - \frac{(G)^2}{H + \lambda} \right)$$

This method significantly decreases computational complexity and is particularly effective in distributed and out-of-core settings, allowing XGBoost to handle large datasets efficiently.

#### 6.8.2 Construction of Candidate Splits

The split finding algorithm in XGBoost is designed to efficiently propose candidate splits by using methods that reduce computational overhead while maintaining statistical accuracy.

This approach allows XGBoost to perform well on large datasets, making it a popular choice for machine learning practitioners.

One key technique is the use of *weighted quantile sketching*, which enables XGBoost to select candidate split points that are statistically representative of the data distribution, rather than exhaustively evaluating all possible splits. By focusing on statistically significant splits, the algorithm achieves a balance between performance and computational efficiency.

### 6.8.2.1 Weighted Quantile Sketch

The *weighted quantile sketch* technique in XGBoost is a method for constructing candidate split points by approximating the distribution of weighted data, which is especially useful for large datasets. This approach ensures that the candidate splits reflect the distribution of the data accurately while reducing computational overhead.

- **Definition:** Weighted quantile sketch is a data structure that approximates quantiles in a dataset where each data point has an associated weight. Unlike standard quantile methods, which assume all instances are equally important, weighted quantile sketch accounts for weights in the dataset, enabling more accurate representation of data distribution in scenarios where instances carry different levels of importance (weights).
- **Rank Function and Candidate Splits:** Let  $\mathcal{D}_k = \{(x_{1k}, h_1), (x_{2k}, h_2), \dots, (x_{nk}, h_n)\}$  be the set of feature values  $x_k$  along with their respective Hessian weights  $h_i$  for each training instance  $i$ . We define the rank function  $r_k(z)$  for a feature value  $z$  as:

$$r_k(z) = \frac{1}{\sum_{(x,h) \in \mathcal{D}_k} h} \sum_{\substack{(x,h) \in \mathcal{D}_k \\ x < z}} h,$$

where  $r_k(z)$  represents the cumulative proportion of instances whose feature value  $x$  for feature  $k$  is smaller than  $z$ . The objective is to find a set of candidate split points  $\{s_{k1}, s_{k2}, \dots, s_{kl}\}$  such that:

$$|r_k(s_{kj}) - r_k(s_{kj+1})| < \epsilon,$$

where  $\epsilon$  is an approximation factor that controls the precision of the split points. This rank-based selection ensures that split candidates are spaced proportionally, reflecting the weighted distribution of the data.

- **Approximation Factor and Candidate Points:** The approximation factor  $\epsilon$  dictates the granularity of the candidate points, resulting in roughly  $1/\epsilon$  candidate points. Each candidate point is weighted by its Hessian value  $h_i$ , representing the significance of the instance within the dataset. This weighted approach allows XGBoost to prioritize split points that better reflect data density and importance.
- **Objective Reformulation:** To incorporate weights effectively, the objective function is reformulated to focus on weighted squared loss. The objective can be approximated

as:

$$\sum_{i=1}^n \frac{1}{2} h_i (f_t(x_i) - g_i/h_i)^2 + \Omega(f_t) + \text{constant},$$

where  $g_i$  is the gradient and  $h_i$  is the Hessian for each instance. This formulation is particularly advantageous for large datasets, as it enables efficient approximation without evaluating each split exhaustively.

- **Distributed and Weighted Setting:** Traditional quantile sketches work well for unweighted data but do not extend naturally to weighted datasets. For datasets with varying weights, naive quantile methods or sampling techniques do not guarantee theoretical accuracy. The weighted quantile sketch in XGBoost addresses this limitation by providing a method that supports merge and prune operations with provable accuracy. This allows for distributed computation with a guaranteed approximation error, making it ideal for large, distributed datasets.
- **Use Case and Advantages:** Weighted quantile sketching is particularly effective in distributed environments, where data may be partitioned across multiple machines. By creating a sketch structure that maintains a consistent approximation, the technique provides an efficient method to compute split candidates without sorting the full dataset. This is especially beneficial for large datasets where sorting or exhaustive search would be computationally prohibitive.

In summary, the weighted quantile sketch in XGBoost efficiently approximates quantiles in large, weighted datasets, enabling the model to propose accurate split candidates even in distributed computing environments. This technique is essential for XGBoost's scalability and efficiency, allowing it to handle datasets with weighted instances while maintaining computational speed and accuracy.

### 6.8.2.2 Histogram-based Split Finding

The histogram-based split finding algorithm in XGBoost discretizes continuous features into a fixed number of bins, rather than evaluating splits on each unique feature value. This method reduces memory usage and speeds up computation by accumulating gradient and Hessian statistics for each bin instead of individual feature values.

- **Definition:** In the histogram-based method, continuous features are discretized by dividing the feature range into bins. Each bin represents a range of feature values, and gradient statistics are aggregated for all values within each bin.
- **Formula:** The binning process can be defined as:

$$\text{Histogram Bin} = [\min(x_i), \max(x_i)] \rightarrow B$$

where  $B$  denotes the bins created from continuous feature values  $x_i$ . For a feature  $x$ , if we divide its range into  $K$  bins, each bin  $b_k$  for  $k = 1, 2, \dots, K$  can accumulate

gradients  $G$  and Hessians  $H$  as:

$$G_{b_k} = \sum_{i \in b_k} g_i, \quad H_{b_k} = \sum_{i \in b_k} h_i$$

where  $g_i$  and  $h_i$  are the gradient and Hessian values for each instance  $i$  in bin  $b_k$ .

- **Use Case:** Histogram-based split finding is ideal for large datasets, as it allows XGBoost to compute splits more efficiently without examining every feature value. This method is especially beneficial in high-dimensional data and when working with sparse data matrices, where many feature values may be zero.

#### 6.8.2.3 Split Gain Calculation

To determine the quality of each candidate split, XGBoost calculates the *split gain*, which represents the reduction in the objective function after performing a split. The split gain formula is as follows:

$$\text{Gain} = \frac{1}{2} \left( \frac{(\sum_{i \in L} g_i)^2}{\sum_{i \in L} h_i + \lambda} + \frac{(\sum_{i \in R} g_i)^2}{\sum_{i \in R} h_i + \lambda} - \frac{(\sum_{i \in L \cup R} g_i)^2}{\sum_{i \in L \cup R} h_i + \lambda} \right) - \gamma$$

where:

- $L$  and  $R$  represent the left and right child nodes after the split.
- $g_i$  and  $h_i$  are the gradient and Hessian values for each instance  $i$ .
- $\lambda$  is the regularization parameter to penalize complex trees.
- $\gamma$  is the minimum loss reduction required to make a further split, controlling the complexity of the tree.

The split gain is used to select the best split point by maximizing the gain, thus ensuring that each split contributes to reducing the overall error of the model.

#### 6.8.2.4 Advantages and Applications

The split finding algorithm in XGBoost, combining histogram-based binning with weighted quantile sketching, is advantageous in multiple scenarios:

- It reduces the computational cost of split finding, making XGBoost highly scalable for large datasets.
- It adapts well to high-dimensional data by focusing on significant splits rather than exhaustive search.

- Weighted quantile sketching enhances performance in distributed settings by approximating split points across partitions.
- Histogram-based binning is especially useful for dense data, allowing efficient gradient and Hessian accumulation within bins.

In summary, XGBoost's split finding algorithm is designed to balance computational efficiency and statistical accuracy, making it highly effective for a wide range of machine learning tasks, particularly those involving large-scale or high-dimensional data.

### 6.8.3 Objective Function

The objective function in XGBoost consists of a differentiable convex loss function that measures the difference between predicted and actual values and a regularization term to penalize model complexity, which helps prevent overfitting.

#### 6.8.3.1 Loss Function

The loss function quantifies the error between the predicted values and the actual target values. It varies based on the problem type.

**For Regression** For regression tasks, the squared error loss function is commonly used:

$$\mathcal{L}(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$$

where  $y_i$  is the actual value, and  $\hat{y}_i$  is the predicted value.

**For Classification** For binary classification, the logistic loss function is utilized:

$$\mathcal{L}(y_i, \hat{y}_i) = -[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

where  $y_i$  is the label (0 or 1), and  $\hat{y}_i$  is the predicted probability.

#### 6.8.3.2 Unregularized XGBoost Algorithm

XGBoost operates using a Newton-Raphson method in function space, as opposed to standard gradient boosting, which uses gradient descent in function space. A second-order Taylor approximation is utilized in the loss function to connect it to the Newton-Raphson method.

---

**Algorithm** Unregularized XGBoost Algorithm

---

**Initialization**

- **Input:** Training set  $\{(x_i, y_i)\}_{i=1}^N$ , a differentiable loss function  $\mathcal{L}(y, F(x))$ , a number of weak learners  $M$ , and a learning rate  $\alpha$ .
- **Initialize** model with a constant value:

$$\hat{f}_{(0)}(x) = \arg \min_{\theta} \sum_{i=1}^N \mathcal{L}(y_i, \theta).$$

**Iterative Process****for**  $m = 1$  to  $M$  **do**

- Compute the *gradients* and *hessians* for each instance:

$$\hat{g}_m(x_i) = \left[ \frac{\partial \mathcal{L}(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x)=f_{(m-1)}(x)},$$

$$\hat{h}_m(x_i) = \left[ \frac{\partial^2 \mathcal{L}(y_i, f(x_i))}{\partial f(x_i)^2} \right]_{f(x)=f_{(m-1)}(x)}.$$

- Fit a base learner (e.g., tree) using the training set  $\left\{x_i, \frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)}\right\}_{i=1}^N$  by solving the optimization problem:

$$\hat{\phi}_m = \arg \min_{\phi \in \Phi} \sum_{i=1}^N \frac{1}{2} \hat{h}_m(x_i) \left[ \phi(x_i) - \frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)} \right]^2.$$

- Update the base learner:

$$f_m(x) = \alpha \hat{\phi}_m(x).$$

- Update the model:

$$f_{(m)}(x) = f_{(m-1)}(x) - f_m(x).$$

**end for****Output**

- The final model:

$$f_{(M)}(x) = \sum_{m=0}^M f_m(x).$$

### 6.8.3.3 Regularization Term

Regularization in XGBoost penalizes complex models by adding a term:

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \sum_j w_j^2$$

where  $T$  is the number of leaves in the tree,  $w_j$  represents the leaf weights,  $\gamma$  is the penalty for adding a new leaf, and  $\lambda$  is the L2 regularization term on leaf weights. This regularization helps control the model's complexity and enhances its generalization capabilities.

### 6.8.4 Sparsity-Aware Splitting Algorithm

The *sparsity-aware splitting algorithm* in XGBoost is designed to efficiently handle missing values and sparse data during split finding. Instead of imputing missing values or processing zero values as regular entries, XGBoost introduces default directions based on the gain from different split scenarios, allowing it to handle sparse datasets without unnecessary computations.

#### 6.8.4.1 Objective of Sparsity-Aware Splitting

In many real-world datasets, certain features may have missing or zero values. The sparsity-aware splitting algorithm in XGBoost accommodates these by:

- Calculating split gains by considering missing values as going in either left or right directions.
- Choosing the split direction for missing values that maximizes the gain, leading to a more efficient and accurate model.

The algorithm works by sorting instances based on feature values, then iteratively computing gains for each split, considering both scenarios where missing values go to the left or the right child node.

#### 6.8.4.2 Pseudocode of Sparsity-Aware Splitting

The pseudocode for the sparsity-aware splitting algorithm is shown below, which handles missing entries during the split decision process.

---

**Algorithm** Sparsity-aware Split Finding

---

**Initialization**

- **Input:**  $I$ , instance set of the current node;  $I_k = \{i \in I \mid x_{ik} \neq \text{missing}\}$ , set of non-missing entries for feature  $k$ ;  $d$ , feature dimension.
- Compute initial sums:  $G \leftarrow \sum_{i \in I} g_i$ ,  $H \leftarrow \sum_{i \in I} h_i$ , gain  $\leftarrow 0$ .

**Iterative Process****for**  $k = 1$  to  $m$  **do** $G_L \leftarrow 0$ ,  $H_L \leftarrow 0$ Consider missing values going to the right**for**  $j$  in sorted( $I_k$ , ascending order by  $x_{jk}$ ) **do** $G_L \leftarrow G_L + g_j$ ,  $H_L \leftarrow H_L + h_j$ ,  $G_R \leftarrow G - G_L$ ,  $H_R \leftarrow H - H_L$  $\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$ **end for**Consider missing values going to the left $G_R \leftarrow 0$ ,  $H_R \leftarrow 0$ **for**  $j$  in sorted( $I_k$ , descending order by  $x_{jk}$ ) **do** $G_R \leftarrow G_R + g_j$ ,  $H_R \leftarrow H_R + h_j$ ,  $G_L \leftarrow G - G_R$ ,  $H_L \leftarrow H - H_R$  $\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$ **end for****end for****Output:** Split and default directions with max gain.

---

**Explanation of the Pseudocode**

- **Inputs:**

- $I$ : The set of instances in the current node.
- $I_k$ : Subset of  $I$  where feature  $x_k$  is not missing for each instance.
- $d$ : The feature dimension under consideration for the split.

- **Initialization:**

- Compute the overall gradient sum  $G = \sum_{i \in I} g_i$  and Hessian sum  $H = \sum_{i \in I} h_i$  for all instances in the node. These represent the total gradients and Hessians, which are used to calculate the overall gain.

- **Loop over each feature  $k$ :** The algorithm iterates over each feature to find the best split.

- **Handling Missing Values Going Right:**

- For each possible split point  $j$  (sorted in ascending order of feature values), accumulate the left and right gradients ( $G_L$ ,  $G_R$ ) and Hessians ( $H_L$ ,  $H_R$ ).

- Calculate the gain score for this split as:

$$\text{score} = \max \left( \text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda} \right),$$

which measures the quality of the split.

- **Handling Missing Values Going Left:**

- Repeat the above process but assume missing values go to the left. The instances are processed in descending order of feature values.
- Update the gain score for each split point as in the previous step.
- **Output:** After processing all features, the algorithm outputs the split and default direction for missing values that yield the maximum gain.

#### 6.8.4.3 Purpose and Advantages

The sparsity-aware splitting algorithm in XGBoost allows efficient handling of missing data during training. Instead of filling in missing values, the algorithm chooses the optimal direction (left or right) for missing entries based on the split gain, resulting in better performance and computational efficiency. This is especially useful for datasets with numerous missing values or sparse matrices, where zero values represent an absence of information rather than a meaningful feature value.

### 6.8.5 Second-order Approximation

XGBoost leverages a second-order Taylor expansion of the objective function, a ***key innovation that allows it to achieve faster convergence and precise updates***. This approximation incorporates both the gradient and the Hessian (second derivative) of the loss function, which provides a more accurate representation of the loss behavior around the current estimate.

#### 6.8.5.1 Taylor Expansion

The Taylor series expansion is used to approximate a function around a given point. For a function  $f(x)$ , the Taylor series expansion around a point  $x_0$  up to the second-order term is:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2$$

where  $f'(x_0)$  is the first derivative (gradient) and  $f''(x_0)$  is the second derivative (Hessian) evaluated at  $x_0$ . This expansion uses both the slope and curvature of  $f(x)$  around  $x_0$  to approximate the function more accurately than using only the gradient.

### 6.8.5.2 Regularized Learning Objective

XGBoost optimizes a regularized objective function at each iteration, defined as:

$$\mathcal{L}(\phi) = \sum_{i=1}^n \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k)$$

where:

- $\mathcal{L}(y_i, \hat{y}_i)$  is the differentiable convex loss function that measures the error between the target  $y_i$  and the prediction  $\hat{y}_i$ .
- $\Omega(f_k) = \gamma T + \frac{1}{2}\lambda\|w\|^2$  is the regularization term that penalizes the complexity of the model. Here,  $T$  represents the number of leaves,  $w$  is the vector of leaf weights,  $\gamma$  controls the number of leaves, and  $\lambda$  controls the  $L2$ -norm of leaf weights.

The regularization term  $\Omega(f_k)$  helps smooth the learned weights to avoid overfitting.

### 6.8.5.3 Second-order Approximation (Taylor Expansion)

The second-order Taylor expansion allows for an efficient approximation of the objective function at each iteration. Given predictions  $\hat{y}_i^{(t-1)}$  at the previous step  $t - 1$ , the new objective function when adding function  $f_t$  is approximated as:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^n \left[ \mathcal{L}(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t)$$

where:

- $g_i = \frac{\partial \mathcal{L}(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}$  is the gradient (first derivative) of the loss function with respect to the prediction.
- $h_i = \frac{\partial^2 \mathcal{L}(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)2}}$  is the Hessian (second derivative) of the loss function with respect to the prediction.

This expansion allows XGBoost to consider both the direction (through  $g_i$ ) and the confidence (through  $h_i$ ) of the update, which provides a more precise adjustment compared to using only first-order gradients.

#### 6.8.5.4 Gain from a Split

To decide the best split at each node, XGBoost calculates the *gain* of the split, which represents the reduction in the objective function. For a dataset split into left  $L$  and right  $R$  subsets, the gain Gain is defined as:

$$\text{Gain} = \frac{1}{2} \left( \frac{(\sum_{i \in L} g_i)^2}{\sum_{i \in L} h_i + \lambda} + \frac{(\sum_{i \in R} g_i)^2}{\sum_{i \in R} h_i + \lambda} - \frac{(\sum_{i \in L \cup R} g_i)^2}{\sum_{i \in L \cup R} h_i + \lambda} \right) - \gamma$$

where:

- $g_i$  and  $h_i$  are the gradient and Hessian of the loss function for each data point  $i$ .
- $\lambda$  is a regularization parameter that penalizes leaf weights, reducing model complexity.
- $\gamma$  is the minimum loss reduction required to make a split. This helps control the depth of the tree and prevent overfitting.

This gain formula helps XGBoost choose splits that provide the maximum reduction in the objective function, thus enhancing the model's accuracy iteratively.

#### 6.8.5.5 Advantages of the Second-order Approximation in XGBoost

By incorporating the second-order term in the Taylor expansion, XGBoost can make more refined updates to the model parameters, which contributes to:

- **Faster Convergence:** More accurate updates allow the model to reach optimal solutions in fewer iterations.
- **Improved Stability:** The inclusion of the Hessian helps stabilize the updates, reducing the chances of oscillations.
- **Enhanced Performance:** With a more precise approximation, XGBoost can achieve higher accuracy with fewer boosting rounds.

### 6.8.6 Hyperparameters

XGBoost provides several hyperparameters that control various aspects of model behavior, such as learning rate, tree depth, and regularization terms. Proper tuning of these hyperparameters is essential for optimal model performance.

- **Learning Rate (eta):**
  - **Definition:** The learning rate, denoted as  $\eta$ , controls the contribution of each tree to the model. A lower learning rate slows down the learning process, often requiring more trees but resulting in better generalization.

- **Typical Range:** Values between 0.01 and 0.3 are commonly used.
- **Effect:** Lower values reduce the risk of overfitting but require more boosting rounds (trees).
- **Tree Depth (max\_depth):**
  - **Definition:** The maximum depth of each tree controls its complexity. Shallow trees prevent overfitting but may underfit, while deeper trees capture more complex patterns at the risk of overfitting.
  - **Typical Range:** Values between 3 and 10 are generally effective, with 6 as a typical starting point.
  - **Effect:** Shallow trees (lower max\_depth) reduce overfitting, while deeper trees allow for more complex interactions at the risk of overfitting.
- **Minimum Child Weight (min\_child\_weight):**
  - **Definition:** Controls the minimum sum of instance weights (Hessian) required in a child node to make a split. Higher values limit small nodes, reducing overfitting.
  - **Typical Range:** Values between 1 and 10.
  - **Effect:** Higher values restrict splits based on fewer data points, thus reducing overfitting.
- **Gamma (min\_split\_loss):**
  - **Definition:** Gamma specifies the minimum reduction in loss function needed to make a split. It controls tree complexity by setting a threshold for splits.
  - **Typical Range:** Values between 0 and 5.
  - **Effect:** Higher gamma values lead to fewer splits, reducing overfitting by simplifying the model.
- **Subsample:**
  - **Definition:** Specifies the fraction of training data to sample for each tree. Lower values introduce randomness, preventing overfitting by training each tree on only a portion of the data.
  - **Typical Range:** Values between 0.5 and 1.
  - **Effect:** Lower values help avoid overfitting by reducing model variance but may cause underfitting if set too low.
- **Column Sampling (colsample\_bytree, colsample\_bylevel, colsample\_bynode):**
  - **Definition:**
    - \* **colsample\_bytree:** Fraction of features sampled for each tree.
    - \* **colsample\_bylevel:** Fraction of features sampled for each level of the tree.

- \* **colsample\_bynode**: Fraction of features sampled for each node.
  - **Typical Range**: Values between 0.5 and 1.
  - **Effect**: Using a subset of features at each split reduces correlation among trees, enhancing generalization and reducing overfitting.
- **Lambda (L2 Regularization)**:
  - **Definition**: The L2 regularization term on leaf weights controls the complexity of the model. It penalizes large weights, reducing overfitting.
  - **Typical Range**: Values between 0 and 10.
  - **Effect**: Higher values make the model more conservative, dampening the impact of individual features.
- **Alpha (L1 Regularization)**:
  - **Definition**: The L1 regularization term on leaf weights encourages sparsity, effectively setting less important weights to zero and improving feature selection.
  - **Typical Range**: Values between 0 and 10.
  - **Effect**: Higher values lead to simpler models, with more weights driven to zero, which can prevent overfitting.
- **Scale\_pos\_weight**:
  - **Definition**: Scales the gradient for positive instances, useful for imbalanced datasets.
  - **Typical Range**: Usually set to num\_negative/num\_positive.
  - **Effect**: Focuses the model on the minority class in imbalanced classification tasks, helping balance prediction accuracy.
- **Number of Boosting Rounds (n\_estimators)**:
  - **Definition**: Specifies the total number of trees to add to the model sequentially.
  - **Typical Range**: Varies significantly based on dataset and learning rate.
  - **Effect**: More trees generally improve accuracy but can increase overfitting risk. Early stopping can be used to determine the optimal number of rounds.
- **Early Stopping**:
  - **Definition**: Stops training when the validation set does not show improvement for a specified number of rounds, preventing overfitting and saving computational resources.
  - **Typical Range**: 10 to 50 rounds are commonly used.
  - **Effect**: Early stopping helps to prevent overfitting by stopping the training when no improvement is observed.

**Hyperparameter Tuning Strategy:** A good strategy for tuning these hyperparameters is:

1. Start with tuning **max\_depth** and **min\_child\_weight**.
2. Adjust **gamma** to control model complexity.
3. Fine-tune **subsample** and **colsample\_bytree** for further regularization.
4. Finally, adjust the **learning rate** and **n\_estimators** for optimal performance.

## 6.8.7 Column Sampling

Column sampling is a technique in XGBoost where a random subset of features is selected at each split or tree construction step. This method helps prevent overfitting and improves the model's ability to generalize by reducing the variance of the learned model.

### 6.8.7.1 Definition

Column sampling, also known as feature subsampling, refers to the process of randomly selecting a subset of features (columns) from the entire feature set. This subset is used to construct each tree or each split within a tree. In XGBoost, users can specify the fraction of features to sample using parameters such as:

- **colsample\_bytree**: Fraction of features to consider for each tree.
- **colsample\_bylevel**: Fraction of features to consider at each level of the tree.
- **colsample\_bynode**: Fraction of features to consider at each node split.

This randomness in feature selection means that each tree in the ensemble may learn slightly different patterns, enhancing the model's diversity.

### 6.8.7.2 Use Case

Column sampling is particularly useful in high-dimensional datasets where the number of features is large, and many of them may be irrelevant or redundant. By sampling a subset of features at each split, XGBoost can:

- Reduce the chance of overfitting on irrelevant features.
- Encourage each tree to capture different aspects of the data, leading to a more robust ensemble.
- Speed up training, as fewer features are considered at each split.

In scenarios like text classification or gene expression data, where feature dimensions can be very high, column sampling is advantageous as it avoids the risk of over-relying on a small set of features and can handle feature interactions more flexibly.

#### 6.8.7.3 Purpose

The main purposes of column sampling in XGBoost are:

- **Reduce Overfitting:** By using only a fraction of features, the model does not overly depend on any single feature. This randomness forces the trees to learn unique patterns, reducing the model's tendency to memorize noise in the training data.
- **Improve Generalization:** The randomness introduced by column sampling helps reduce variance. Since each tree only sees a subset of features, it becomes less likely for the ensemble model to be biased towards specific features, enhancing the generalization capabilities of the model on unseen data.
- **Increase Computational Efficiency:** When only a subset of features is considered at each split, the computational burden of searching for the best split across all features is reduced, leading to faster training times, especially with high-dimensional data.

Column sampling in XGBoost is inspired by techniques used in Random Forests, where feature subsampling at each split is a key factor contributing to the ensemble's ability to avoid overfitting. The balance between selecting relevant features and introducing randomness is crucial for achieving a model that is both accurate and generalizable.

### 6.8.8 Parallelization

XGBoost incorporates parallelization techniques to speed up training on large datasets by taking advantage of multiple cores or distributed systems. Parallelization is crucial in XGBoost because it helps to reduce the time required for calculating gradients, finding optimal split points, and constructing trees.

#### 6.8.8.1 Building Trees Level-wise

In XGBoost, trees are built level-wise rather than leaf-wise, which means all nodes at a given depth are split before moving to the next level. This structure allows for parallel computations, as the potential splits for all nodes at a given depth can be computed simultaneously. By doing so, XGBoost can:

- Reduce computation time, as each level is constructed concurrently.
- Improve memory usage efficiency by processing all nodes at a particular level before moving down the tree.

- Enhance model interpretability by ensuring balanced trees, as opposed to building very deep nodes one at a time.

This level-wise approach is particularly beneficial for handling wide datasets with many features, where each feature's split can be evaluated independently at each level.

#### 6.8.8.2 Parallel Computations

XGBoost performs parallel computation primarily by splitting the gradient and Hessian calculations among multiple processing threads. The main areas where parallelization is applied are:

- **Gradient and Hessian Computations:** XGBoost calculates the gradient and Hessian values for each instance in the dataset, which are used to find the optimal splits. These computations are distributed across multiple threads, significantly speeding up the calculation of gradients and Hessians.
- **Split Finding:** For each feature, potential split points are evaluated in parallel to determine the one that maximizes the gain in the objective function. This includes calculating sums of gradients and Hessians for left and right nodes for each potential split point, which can be independently processed for different features.
- **Column Block Splitting:** XGBoost divides features into blocks so that they can be processed concurrently. This columnar approach ensures that each thread focuses on a subset of features, further accelerating the split-finding process.

By distributing these computations across multiple threads or machines, XGBoost can significantly reduce training time, especially for large datasets. This parallelization strategy is flexible and adapts to both multi-core processors and distributed computing environments.

### 6.8.9 Scalability

XGBoost is designed to handle very large datasets efficiently, making it suitable for applications in big data and large-scale machine learning tasks. Scalability in XGBoost is achieved through techniques that allow it to operate effectively when datasets are too large to fit into memory or when distributed computing resources are available.

#### 6.8.9.1 Out-of-core Computing

Out-of-core computing enables XGBoost to process datasets that exceed the system's available memory by utilizing disk storage. This is achieved by:

- **Data Sharding:** Large datasets are split into smaller chunks, or shards, which are processed one at a time. Each shard fits into memory, while the rest are stored on disk.

- **Pre-fetching:** As one data shard is processed, the next shard is loaded from disk into memory, minimizing the I/O wait time. This pre-fetching approach helps maintain computational efficiency by overlapping I/O operations with processing.
- **Incremental Updates:** As each shard is processed, the model parameters are updated incrementally, allowing XGBoost to effectively learn from large datasets without requiring the entire dataset to be loaded into memory at once.

Out-of-core computing is particularly useful for scenarios where memory resources are limited, such as on commodity hardware, while maintaining high performance on large datasets.

#### 6.8.9.2 Distributed Computing

XGBoost supports distributed computing, allowing the model to be trained across multiple machines in a cluster, which makes it highly scalable for large datasets. Distributed computing in XGBoost involves:

- **Data Partitioning:** The dataset is divided across multiple machines, with each machine handling a subset of the data. This reduces the memory and computational load on each individual machine.
- **Synchronous Gradient Updates:** Each machine calculates gradient and Hessian values for its data subset. These values are then synchronized across machines to ensure consistent updates of the model parameters.
- **Parameter Averaging and Aggregation:** After each iteration, the gradient and Hessian statistics are aggregated across all machines to compute the optimal split points. The model parameters are then updated in a synchronized manner across the cluster.
- **Fault Tolerance:** XGBoost can recover from failures in distributed settings, which enhances reliability in large-scale applications.

This distributed approach allows XGBoost to scale horizontally by adding more machines to the cluster, making it suitable for big data applications. It leverages frameworks like Apache Spark, Hadoop, and Dask for seamless integration in distributed environments, enabling it to handle billions of examples efficiently.

## 6.9 Distinguishing between ensembling, hybrid, mixing, and mutating

### 6.9.1 Ensembling

**Definition:** Ensembling involves using a collection of individual models to make predictions. Each model in the ensemble typically captures different patterns or aspects of the data, and their predictions are combined to create a more robust final prediction.

**Methods:**

- **Bootstrap Aggregating (often called Bagging):** Multiple models are trained on different subsets of the training data that are sampled with replacement. This technique reduces model variance and is widely used in models such as Random Forests.
- **Boosting:** Models are trained sequentially, where each model tries to correct the mistakes made by the previous model in the sequence. This approach reduces bias and is used in models like Adaptive Boosting and Gradient Boosting.
- **Stacking:** Predictions from several models are used as input features for another model (called a meta-learner) which then makes the final prediction.

**Goal:** Ensembling aims to reduce the variance and bias of predictions, improving accuracy and robustness by aggregating multiple model outputs.

### 6.9.2 Hybrid Models

**Definition:** Hybrid models combine different types of models, for instance, by blending a neural network with a decision tree or combining a rule-based system with a machine learning model. This approach leverages different model characteristics to enhance performance.

**Example:** An example of a hybrid model could be a system that uses a time series analysis model alongside a long short-term memory network (which is a type of neural network suitable for time-dependent data) to improve predictions in time-series forecasting.

**Goal:** The main purpose of hybrid models is to leverage the strengths of each model type, such as using a model that is interpretable (like a rule-based system) along with a highly flexible model (like a neural network), to achieve a balance between interpretability and accuracy.

### 6.9.3 Mixing

**Definition:** Mixing refers to combining different types of data representations, features, or even distinct algorithms within a single model. This concept is broader and can apply to any scenario where diverse elements are integrated into a unified model framework.

**Example:** An example of mixing is creating a model that accepts various forms of data (such as text, images, and audio) simultaneously, known as a multi-modal model. Another instance of mixing could involve using different types of neural network layers with diverse activation functions within one model structure.

**Goal:** Mixing aims to improve the generalization capabilities of a model by allowing it to handle complex, multi-modal input data or integrate a wide range of data sources for better overall prediction accuracy.

#### 6.9.4 Mutating

**Definition:** In machine learning, mutating typically refers to making slight, random modifications to model parameters or the model structure. This approach is common in evolutionary algorithms and aims to help explore a broader solution space by introducing variation.

**Example:** Within genetic algorithms, mutation can randomly alter certain parameters of a candidate model. This variation allows the algorithm to avoid getting trapped in local solutions and enables exploration of new potential solutions.

**Goal:** The objective of mutating is to foster diversity in candidate models, enhancing exploration of the solution space and helping avoid premature convergence to suboptimal solutions, which can also help in preventing overfitting.

### Recognition

- **Ensembling** aggregates predictions from multiple models to achieve improved performance.
- **Hybrid Models** combine distinct model types to leverage complementary strengths.
- **Mixing** incorporates varied inputs, data types, or architectural elements in a single model.
- **Mutating** introduces variations in model parameters or structures to broaden the solution search space.

# Chapter 7

## Deep Learning

### 7.1 Neural Networks Notations

**General comments:**

- Superscript ( $i$ ) will denote the  $i^{\text{th}}$  training example while superscript [ $l$ ] will denote the  $l^{\text{th}}$  layer.

**Sizes:**

- $m$ : number of examples in the dataset
- $n_x$ : input size
- $n_y$ : output size (or number of classes)
- $n_h^{[l]}$ : number of hidden units of the  $l^{\text{th}}$  layer
  - In a for loop, it is possible to denote  $n_x = n_h^{[0]}$  and  $n_y = n_h^{[\text{number of layers}+1]}$
- $L$ : number of layers in the network

**Objects:**

- $X \in \mathbb{R}^{n_x \times m}$  is the input matrix
- $x^{(i)} \in \mathbb{R}^{n_x}$  is the  $i^{\text{th}}$  example represented as a column vector
- $Y \in \mathbb{R}^{n_y \times m}$  is the label matrix
- $y^{(i)} \in \mathbb{R}^{n_y}$  is the output label for the  $i^{\text{th}}$  example
- $W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in previous layer}}$  is the weight matrix. Super-script [ $l$ ] indicates the layer

- $b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$  is the bias vector in the  $l^{\text{th}}$  layer
- $\hat{y} \in \mathbb{R}^{n_y}$  is the predicted output vector. It can also be denoted  $a^{[L]}$  where  $L$  is the number of layers in the network.

**Common forward propagation equation examples:**

- $a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$  where  $g^{[l]}$  denotes the  $l^{\text{th}}$  layer activation function
- $\hat{y}^{(i)} = \text{softmax}(W_h h + b_2)$
- General Activation Formula:  $a_j^{[l]} = g^{[l]} \left( \sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]} \right) = g^{[l]}(z_j^{[l]})$

## 7.2 Activation Functions

### 7.2.1 Introduction and Historical Background

In artificial neural networks, an **activation function** is a mathematical function that determines the output of a neuron based on its inputs and weights. It plays a crucial role in the architecture of neural networks by introducing non-linearity into the model, allowing it to capture complex relationships and patterns in the data. Without activation functions, neural networks would behave as simple linear models, significantly limiting their ability to model non-linear data.

Historically, early neural networks were inspired by the structure of the human brain, where neurons fire only when their inputs exceed a certain threshold. The earliest artificial neurons used a simple **binary step function** as the activation function, where the output was either 0 or 1 based on whether the input surpassed a certain threshold. While this approach worked for simple tasks, it was insufficient for handling more complex, non-linear relationships.

### 7.2.2 Why Activation Functions Are Needed

In a neural network without activation functions, each layer can only perform linear transformations on its input. Mathematically, this can be expressed as:

$$y = W_1 W_2 \cdots W_n x + b,$$

where  $W_i$  are weight matrices and  $b$  is a bias term. Regardless of the number of layers, the composition of multiple linear transformations remains linear. As a result, a deep network without activation functions would effectively behave as a single-layer linear model, severely limiting its capacity to approximate complex functions.

To overcome this limitation, non-linear activation functions are introduced, allowing the network to approximate non-linear mappings. This enables the network to learn from complex data distributions, making it more powerful and versatile.

### 7.2.3 Importance of Non-Linearity

The ability of neural networks to approximate any continuous function, as stated by the **Universal Approximation Theorem**, relies on the use of non-linear activation functions. A neural network with at least one hidden layer and non-linear activations can act as a universal function approximator, capable of modeling highly complex, non-linear relationships.

If all activation functions were linear (e.g., identity functions), the entire network would reduce to a single linear transformation, regardless of its depth:

$$y = W_1 W_2 \cdots W_n x + b.$$

Introducing non-linearity through activation functions allows the network to capture complex patterns in data, making deep learning effective for tasks like image recognition, speech processing, and natural language understanding.

### 7.2.4 Key Mathematical Properties of Activation Functions

Activation functions possess specific mathematical properties that impact the training and performance of neural networks:

- **Non-linearity:** Non-linear activation functions enable neural networks to model complex, non-linear relationships in data. Without non-linearity, a multi-layer network would reduce to a single linear layer, limiting its expressiveness.
- **Range:** Activation functions can have finite ranges (e.g., Sigmoid:  $(0, 1)$ , Tanh:  $(-1, 1)$ ), which can help stabilize gradient-based training. Others, like the Rectified Linear Unit (ReLU), have an infinite range  $[0, \infty)$ , allowing for more efficient learning but requiring careful tuning to avoid gradient-related issues.
- **Continuously Differentiable:** Differentiability is crucial for efficient gradient-based optimization. While many activation functions, such as Sigmoid and Tanh, are continuously differentiable, ReLU is not differentiable at zero but remains highly effective in practice due to its simplicity and ability to avoid the vanishing gradient problem.
- **Saturating vs. Non-Saturating:** An activation function  $f$  is said to be saturating if:

$$\lim_{|v| \rightarrow \infty} |\nabla f(v)| = 0.$$

Saturating functions like Sigmoid and Tanh can suffer from the *vanishing gradient problem*, making training deeper networks challenging. In contrast, non-saturating functions such as ReLU are preferred for deeper networks because they help maintain non-zero gradients, thereby improving convergence.

## 7.2.5 Types of Activation Functions

Activation functions can be broadly categorized into three types: Ridge functions, Radial functions, and Fold functions. Each of these categories serves different purposes in neural network architectures.

### 7.2.5.1 Ridge Activation Functions

In the context of mathematics and neural networks, **ridge functions** are multivariate functions that can be expressed as the composition of a univariate function with an affine transformation. Formally, a ridge function is defined as:

$$f(\mathbf{x}) = g(\mathbf{x} \cdot \mathbf{a})$$

where:

- $\mathbf{x} \in \mathbb{R}^d$  is the input vector.
- $g : \mathbb{R} \rightarrow \mathbb{R}$  is a univariate function.
- $\mathbf{a} \in \mathbb{R}^d$  is a fixed vector that defines the direction.

The term *ridge function* was coined by B.F. Logan and L.A. Shepp, highlighting its significance in fields such as signal processing and high-dimensional data analysis.

### Key Properties and Relevance

One of the most notable properties of ridge functions is that they are invariant in directions orthogonal to the vector  $\mathbf{a}$ . Specifically, if we take  $d - 1$  independent vectors  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_{d-1}$  that are orthogonal to  $\mathbf{a}$ , any shift of the input vector  $\mathbf{x}$  along these orthogonal directions does not affect the value of the function:

$$f\left(\mathbf{x} + \sum_{k=1}^{d-1} c_k \mathbf{a}_k\right) = g\left((\mathbf{x} + \sum_{k=1}^{d-1} c_k \mathbf{a}_k) \cdot \mathbf{a}\right) = g(\mathbf{x} \cdot \mathbf{a}) = f(\mathbf{x})$$

This implies that **ridge functions are constant in  $d - 1$  directions** perpendicular to  $\mathbf{a}$ . As a result, they are less prone to the curse of dimensionality, making them particularly useful in high-dimensional estimation problems.

### Application in Neural Networks

In neural networks, ridge functions are employed as **activation functions** to introduce non-linearity and allow the model to capture complex patterns. The ridge structure allows these activation functions to effectively reduce the dimensionality of the input data, focusing on essential features along a specific direction  $\mathbf{a}$ . This makes ridge functions particularly useful in tasks such as:

- **Projection Pursuit:** Simplifying high-dimensional data by projecting it onto lower-dimensional spaces.
- **Generalized Linear Models (GLMs):** Extending linear models by introducing non-linearity through ridge functions.
- **Neural Network Activation Functions:** Enabling the network to learn complex, non-linear relationships between inputs and outputs.

By leveraging the directional invariance of ridge functions, neural networks can efficiently learn and generalize from high-dimensional data, especially when combined with other techniques such as regularization and feature selection.

### **Linear Activation Function**

The **Linear activation function** is defined as:

$$\phi(x) = a + x'b$$

**Description:** The linear function outputs a value directly proportional to the input. It is typically used in the output layer for regression tasks.

**Pros:**

- Simple and easy to implement.
- Useful for regression tasks.

**Cons:**

- Cannot handle non-linear relationships.
- Limited expressiveness when used in hidden layers.

**Popular Use Case:** Used in the output layer of regression models.

### **Rectified Linear Unit (ReLU)**

**Historical Context:** ReLU was first used by Alston Householder in 1941 as a mathematical abstraction of biological neural networks. Kunihiko Fukushima reintroduced it in 1969 in the context of visual feature extraction in hierarchical neural networks. Its popularity surged in 2011 when it was shown to enable training of deep supervised neural networks without the need for unsupervised pre-training, unlike the sigmoid and hyperbolic tangent activations that were commonly used before.

The **ReLU function** is defined as:

$$\text{ReLU}(x) = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

where  $x$  is the input to a neuron. This is also known as a ramp function and is analogous to half-wave rectification in electrical engineering.

**Description:** The ReLU function outputs the input directly if it is positive; otherwise, it outputs zero. It has become one of the most popular activation functions in neural networks due to its simplicity and efficiency. The ReLU function is widely used in various applications, including computer vision, speech recognition, and deep learning models, as it enables efficient training of deep networks.

**Pros:**

- **Sparse Activation:** In a randomly initialized network, only about 50% of neurons are activated (i.e., have non-zero output), which leads to sparse representations.
- **Better Gradient Propagation:** Reduces the vanishing gradient problem that occurs with sigmoid and tanh functions.
- **Computational Efficiency:** Requires only a simple comparison and addition operation.
- **Scale-Invariance:** The function is homogeneous, i.e.,  $\text{ReLU}(ax) = a\text{ReLU}(x)$  for  $a \geq 0$ .

**Cons:**

- **Non-Differentiability at Zero:** The derivative is not defined at  $x = 0$ , though in practice, it can be set arbitrarily to either 0 or 1.
- **Dying ReLU Problem:** Neurons can become inactive and stop learning if they enter a state where they always output zero, particularly when a high learning rate is used. This can result in a significant portion of neurons becoming permanently inactive.
- **Non-Zero Centered Output:** Outputs are always non-negative, which can slow down the convergence during training. Batch normalization can help mitigate this issue.
- **Unbounded Output:** Can produce arbitrarily large values, potentially leading to instability during training.

**Variants of ReLU:**

- **Leaky ReLU:**

- **Description:** Leaky ReLU introduces a small positive slope ( $\alpha$ ) for negative inputs to prevent the *dying ReLU* problem, where neurons can become inactive and stop learning if they always output zero.

- **Definition:**

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases}$$

Typically,  $\alpha$  is set to a small value such as 0.01.

- **Difference from ReLU:** Unlike standard ReLU, which outputs zero for all negative inputs, Leaky ReLU assigns a small non-zero gradient to negative inputs, allowing the model to learn even when neurons receive negative inputs.
- **Pros:** Mitigates the dying ReLU problem, especially in deeper networks where neurons may become inactive.
- **Cons:** Introduces an additional hyperparameter ( $\alpha$ ) that needs to be tuned.

- **Parametric ReLU (PReLU):**

- **Description:** PReLU generalizes Leaky ReLU by making the slope  $\alpha$  a learnable parameter instead of being fixed.

- **Definition:**

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases}$$

where  $\alpha$  is learned during training.

- **Difference from Leaky ReLU:** While Leaky ReLU uses a fixed  $\alpha$ , PReLU allows the model to optimize  $\alpha$  for better performance.
- **Pros:** Offers greater flexibility and can adapt to different datasets and architectures.
- **Cons:** Increases the complexity of the model and may lead to overfitting if not regularized properly.

- **Exponential Linear Unit (ELU):**

- **Description:** ELU aims to address bias shifts and improve learning by allowing negative outputs. It introduces exponential decay for negative inputs, which smooths the gradient and helps the network converge faster.

- **Definition:**

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{if } x \leq 0 \end{cases}$$

where  $\alpha$  is a positive hyperparameter.

- **Difference from ReLU:** Unlike ReLU, which outputs zero for negative inputs, ELU outputs a smooth negative value, reducing bias shifts and making the activations zero-centered.
- **Pros:** Helps networks converge faster and achieve higher accuracy by producing zero-centered outputs.
- **Cons:** Requires additional computation due to the exponential function.

- **Softplus:**

- **Description:** Softplus is a smooth approximation to ReLU that eliminates the non-differentiability at zero, producing a continuous and differentiable function.

- **Definition:**

$$f(x) = \ln(1 + e^x), \quad f'(x) = \frac{e^x}{1 + e^x}$$

- **Difference from ReLU:** Unlike ReLU, which has a sharp transition at zero, Softplus is a smooth function, making it differentiable everywhere.

- **Pros:** Provides smooth gradients, reducing potential issues with optimization.

- **Cons:** Computationally more expensive than ReLU due to the logarithm and exponential operations.

- **Gaussian Error Linear Unit (GELU):**

- **Description:** GELU is a smooth activation function often used in modern architectures like transformers. It weights inputs based on their value using the cumulative distribution function of the standard normal distribution.

- **Definition:**

$$f(x) = x\Phi(x)$$

where  $\Phi(x)$  is the cumulative distribution function of the standard normal distribution.

- **Difference from ReLU:** While ReLU simply thresholds negative inputs to zero, GELU uses a probabilistic approach to decide how much of the input passes through, making it smoother and more adaptable.

- **Pros:** Improves performance in tasks requiring smooth gradient propagation, such as natural language processing.

- **Cons:** More computationally intensive due to the use of the cumulative distribution function.

- **SiLU (Swish):**

- **Description:** Also known as the Swish function, SiLU is a smooth, self-gating activation function that outperforms ReLU in some scenarios. It was first introduced in the context of the GELU paper.

- **Definition:**

$$f(x) = x \cdot \text{sigmoid}(x)$$

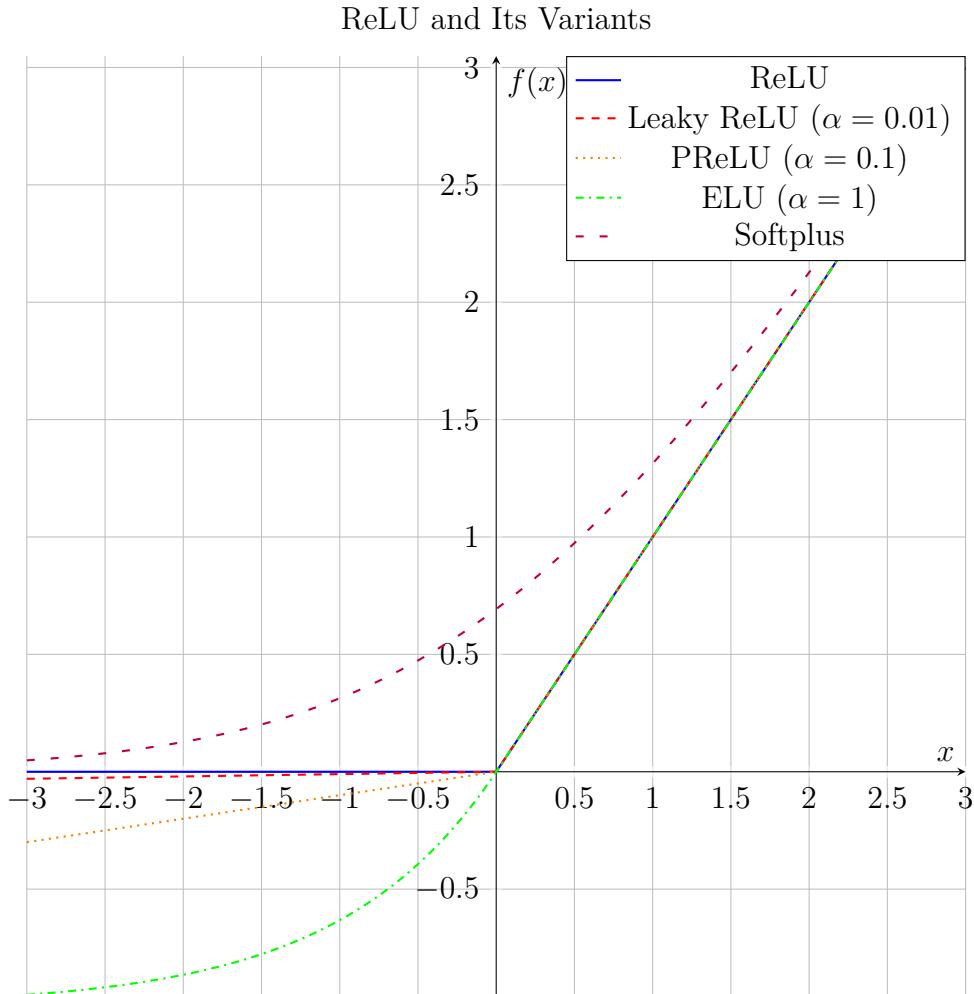
$$\text{where } \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}.$$

- **Difference from ReLU:** Unlike ReLU, which thresholds negative inputs to zero, SiLU allows small negative values, which can improve gradient flow.

- **Pros:** Offers smooth gradients, leading to better performance in deeper networks.

- **Cons:** Computationally more expensive than ReLU.

## ReLU and Its Variants



### Popular Use Cases:

- Widely used in convolutional neural networks (CNNs) for computer vision tasks.
- Effective in feedforward networks for various deep learning applications.
- Has been instrumental in enabling deep learning breakthroughs in fields such as image classification, object detection, and natural language processing.

### Heaviside Step Function

The **Heaviside step function**, also known as the **unit step function**, is commonly denoted by  $H(x)$  or  $\theta(x)$ , and sometimes by  $u(x)$  or  $\mathbf{1}(x)$ . It is a step function named after Oliver Heaviside, which takes the value zero for negative inputs and one for non-negative inputs.

The Heaviside function is widely used in fields such as control systems, signal processing, and theoretical models of neurons, as it can represent a binary switch or on-off signal.

### Definition

The Heaviside step function can be defined in several ways, with different conventions for its value at  $x = 0$ :

$$H(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases}$$

An alternative convention sets  $H(0) = \frac{1}{2}$ , which can be expressed as:

$$H(x) = \frac{1}{2} (1 + \operatorname{sgn}(x))$$

### Other Definitions

The Heaviside function can also be defined using different notations:

- **Iverson Bracket Notation:**

$$H(x) = [x \geq 0]$$

where the bracket evaluates to 1 if the condition is true and 0 otherwise.

- **Indicator Function:**

$$H(x) = \mathbf{1}_{\mathbb{R}_+}(x)$$

where  $\mathbf{1}_{\mathbb{R}_+}(x)$  indicates whether  $x$  is non-negative.

- **As a Limit of the Sigmoid Function:**

$$H(x) = \lim_{k \rightarrow \infty} \frac{1}{1 + e^{-2kx}}$$

- **As a Smooth Approximation Using the Hyperbolic Tangent:**

$$H(x) \approx \frac{1}{2} + \frac{1}{2} \tanh(kx)$$

for large values of  $k$ , which sharpens the transition.

- **Using the Arctangent Function:**

$$H(x) = \lim_{k \rightarrow \infty} \left( \frac{1}{2} + \frac{1}{\pi} \arctan(kx) \right)$$

- **Using the Error Function:**

$$H(x) = \lim_{k \rightarrow \infty} \left( \frac{1}{2} + \frac{1}{2} \operatorname{erf}(kx) \right)$$

## Overview of the Dirac Delta Function

The **Dirac delta function**  $\delta(x)$ , also known as the *unit impulse function*, is a generalized function that is zero everywhere except at  $x = 0$ , where it is infinitely large in such a way that its integral over the entire real line is equal to one:

$$\int_{-\infty}^{\infty} \delta(x) dx = 1.$$

Heuristically, the Dirac delta function can be thought of as a mathematical representation of an "infinitely tall and infinitely narrow" spike centered at zero. It is used in physics and engineering to model instantaneous impulses, such as a sudden force applied at a single point in time.

The Dirac delta function can be defined as the limit of a sequence of functions that are increasingly peaked at  $x = 0$  while maintaining an area of 1 under the curve:

$$\delta(x) = \lim_{a \rightarrow 0} \frac{1}{\sqrt{\pi}a} e^{-\frac{x^2}{a^2}}.$$

## Relationship of the Heaviside Step Function to Dirac delta function

The Dirac delta function is closely related to the **Heaviside step function**  $H(x)$ . In fact, the Dirac delta function  $\delta(x)$  can be considered the weak derivative of the Heaviside function:

$$\delta(x) = \frac{d}{dx} H(x).$$

This means that the delta function represents an instantaneous "spike" in change at  $x = 0$ . Conversely, the Heaviside function can be thought of as the integral of the Dirac delta function:

$$H(x) = \int_{-\infty}^x \delta(s) ds.$$

In practical applications, the Heaviside function models a signal that switches on at  $x = 0$  and remains on indefinitely, while the Dirac delta function is used to represent a sudden, instantaneous event or impulse at a specific point.

## Intuition and Applications

The relationship between these two functions can be understood as follows:

- The Heaviside function  $H(x)$  "steps" from 0 to 1 at  $x = 0$ .
- The Dirac delta function  $\delta(x)$  represents the instantaneous change in the value of  $H(x)$  at  $x = 0$ .

These functions are widely used in fields like control systems, signal processing, and theoretical physics. For example:

- The Dirac delta function is used to model point charges, impulses, and idealized collisions.
- The Heaviside step function is often used in systems where a signal needs to be turned on or off at a specific time.

## Analytic Approximations

Smooth approximations of the Heaviside step function are often used in applications where a non-differentiable function is not suitable. For instance:

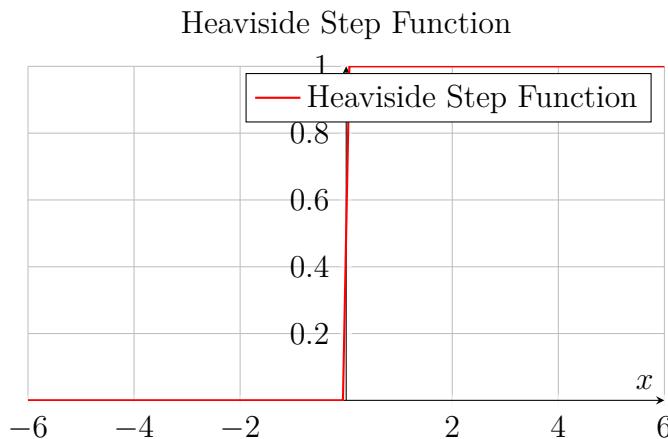
$$H(x) \approx \frac{1}{2} + \frac{1}{2} \tanh(kx) = \frac{1}{1 + e^{-2kx}}$$

where a larger  $k$  results in a sharper transition at  $x = 0$ . The limit of this expression as  $k \rightarrow \infty$  gives the exact step function.

## Applications

The Heaviside function is used in:

- **Signal Processing:** To represent signals that switch on or off at a certain point in time.
- **Control Systems:** As a model for control signals that switch states.
- **Theoretical Neuroscience:** In early models of neurons to represent binary firing states.
- **Differential Equations:** For solving problems involving step inputs or discontinuities.



## Pros

- Simple to implement and understand.
- Useful for binary decision-making and modeling on-off switches.

## Cons

- Not differentiable at  $x = 0$ , making it unsuitable for gradient-based optimization.
- Binary output limits its use in modeling complex, continuous relationships.

### 7.2.5.2 Radial Activation Functions

Radial activation functions are a special class of activation functions where the output value is determined based on the distance of the input vector from a center point  $\mathbf{c}$ . These functions are widely used in Radial Basis Function (RBF) networks, where they help model complex, non-linear relationships by measuring how close the input is to a given center. RBF networks leverage these functions in various tasks such as function approximation, pattern recognition, and clustering.

#### Gaussian Function

The **Gaussian function** is one of the most commonly used radial basis functions. It is defined as:

$$\varphi(\mathbf{v}) = \exp\left(-\frac{\|\mathbf{v} - \mathbf{c}\|^2}{2\sigma^2}\right),$$

where:

- $\mathbf{v}$  is the input vector.
- $\mathbf{c}$  is the center vector.
- $\sigma$  is a parameter that controls the spread of the function.

**Description:** The Gaussian function outputs values that decrease exponentially as the input  $\mathbf{v}$  moves away from the center  $\mathbf{c}$ . The parameter  $\sigma$  determines how rapidly the output value falls off. A smaller  $\sigma$  results in a narrower peak around the center, while a larger  $\sigma$  creates a wider, flatter peak. This function is widely used in RBF networks for tasks like clustering and interpolation due to its smooth, bell-shaped curve.

#### Multiquadric Function

The **Multiquadric function** is defined as:

$$\varphi(\mathbf{v}) = \sqrt{\|\mathbf{v} - \mathbf{c}\|^2 + a^2},$$

where  $a$  is a shape parameter that controls the curvature of the function.

**Description:** This function increases with the distance from the center  $\mathbf{c}$ , making it useful in cases where higher distances should result in higher activation values. It is typically used in surface fitting and interpolation tasks.

#### Inverse Multiquadric Function

The **Inverse Multiquadric function** is defined as:

$$\varphi(\mathbf{v}) = \frac{1}{(\|\mathbf{v} - \mathbf{c}\|^2 + a^2)^{-\frac{1}{2}}},$$

where  $a$  is a parameter that affects the smoothness of the function.

**Description:** Unlike the multiquadric function, the inverse multiquadric decreases as the distance increases. It has a global influence, making it suitable for approximating smooth surfaces.

### Polyharmonic Splines

Polyharmonic splines are defined based on the power of the distance. They come in two forms:

$$\varphi(\mathbf{v}) = \begin{cases} \|\mathbf{v} - \mathbf{c}\|^k, & k = 1, 3, 5, \dots \\ \|\mathbf{v} - \mathbf{c}\|^k \ln(\|\mathbf{v} - \mathbf{c}\|), & k = 2, 4, 6, \dots \end{cases}$$

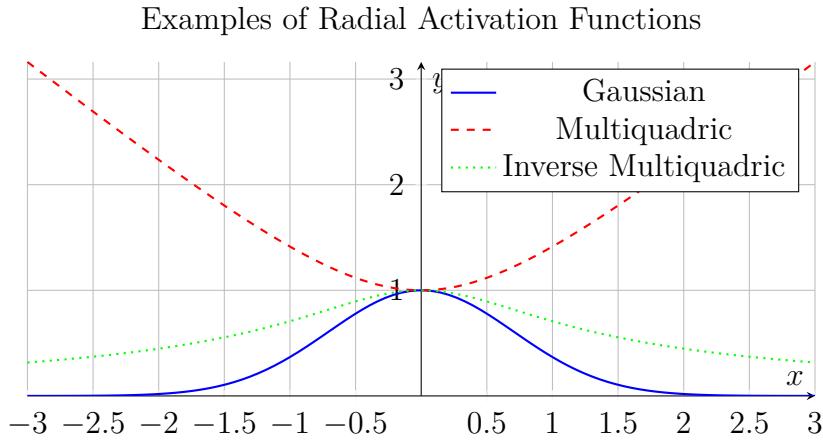
**Description:** These functions are commonly used in interpolation problems. They are particularly useful when exact interpolation of the data is required, especially in scattered data interpolation.

### Applications of Radial Activation Functions

Radial activation functions are extensively used in machine learning models such as RBF networks. The primary advantage of using radial functions lies in their ability to model localized, non-linear phenomena, making them ideal for:

- Function approximation: RBFs can approximate any continuous function given enough basis functions.
- Classification tasks: Due to their localization properties, they are effective in distinguishing classes based on the distance to predefined centers.
- Image reconstruction and surface fitting: In fields such as computer graphics and geostatistics, radial basis functions help in interpolating surfaces from scattered data points.

**Note:** RBFs can also be utilized as kernels in support vector machines (SVMs) to separate non-linearly separable data by mapping it into a higher-dimensional space.



### 7.2.5.3 Fold Activation Functions

In the context of neural networks, the concept of **Fold Activation Functions** is closely related to the idea of accumulating or reducing information over a sequence. The term "fold" in this context is inspired by its origins in functional programming, particularly in languages like Haskell, where fold functions are used to systematically process and aggregate elements in data structures. However, in the context of neural networks, fold activation functions refer to functions that "compress" or "squeeze" outputs to a bounded range, often for the purposes of normalizing outputs or introducing non-linearity.

#### Relation to Fold Functions in Functional Programming

In functional programming, a **fold** (or reduce) function systematically traverses a data structure (like a list) to combine its elements into a single value using a specified combining function. For example, in Haskell:

```
foldr (+) 0 [1, 2, 3, 4] -- Result: 10
```

This example sums up the list by folding it from the right. Similarly, fold activation functions in neural networks compress input values into a bounded range, effectively reducing or normalizing the output. This behavior is particularly useful in cases where outputs need to be constrained, such as in classification tasks where probabilities must sum to 1.

#### Common Fold Activation Functions

Two of the most popular activation functions that can be considered "fold" functions due to their compressing behavior are the **Sigmoid** and **Tanh** functions.

#### Sigmoid Function

The **Sigmoid function** is defined as:

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

**Description:** The sigmoid function maps any real-valued number to a range between 0 and 1. This property makes it particularly useful for binary classification tasks where the output

is interpreted as a probability. The function smoothly "compresses" the output, effectively folding an unbounded input range into the interval  $[0, 1]$ .

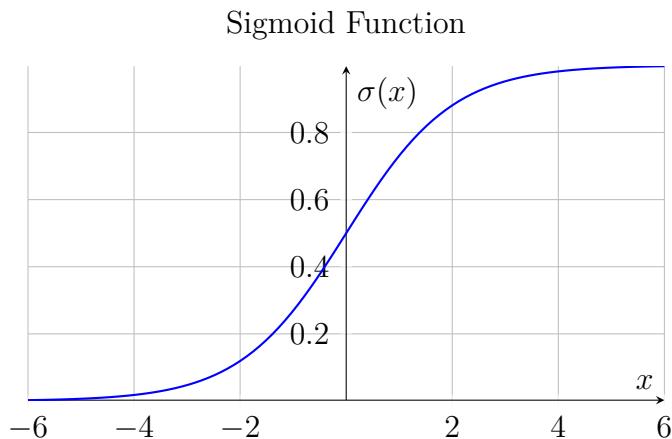
### Pros:

- Smooth and differentiable everywhere, which is beneficial for gradient-based optimization.
- Outputs a probability, making it ideal for binary classification tasks.

### Cons:

- Suffers from the **vanishing gradient problem**, where gradients become very small for extreme values of  $x$ , slowing down learning.
- Not zero-centered, which can lead to slower convergence during training.

**Popular Use Case:** Commonly used in binary classification models, such as logistic regression and binary classification neural networks.



### Tanh Function

The **Tanh function** is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

**Description:** The Tanh function outputs values between  $-1$  and  $1$ , making it zero-centered. This property is advantageous as it can help the model converge faster during training. The tanh function can be seen as a rescaled version of the sigmoid function, effectively "folding" inputs into the range  $[-1, 1]$ .

### Pros:

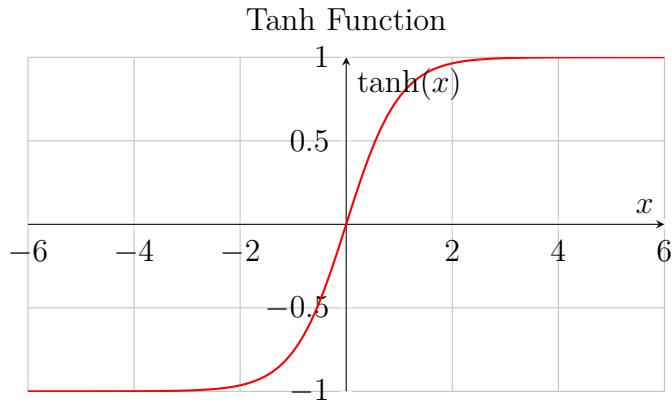
- Zero-centered, leading to faster convergence in gradient descent optimizations.

- Smooth and differentiable, making it suitable for backpropagation.

**Cons:**

- Suffers from the **vanishing gradient problem** similar to the sigmoid function.
- Computationally more expensive than the ReLU function.

**Popular Use Case:** Widely used in Recurrent Neural Networks (RNNs) and LSTMs, where zero-centered activation functions help stabilize training.



### Comparison with Fold Functions in Functional Programming

The concept of "fold" in neural network activation functions is analogous to folding operations in functional programming in the sense that both involve transforming or reducing inputs systematically. However, while functional programming folds are typically used to aggregate data into a single value, fold activation functions compress or normalize neural network outputs, making them bounded within specific ranges.

### Summary

- Fold activation functions, such as Sigmoid and Tanh, compress input values to bounded ranges, enabling efficient optimization and classification.
- The concept of "folding" relates to reducing or normalizing outputs, similar to how fold functions in functional programming combine elements of a data structure.
- Understanding the behavior of these functions is crucial for designing neural network architectures, especially in applications involving sequential data and probabilistic outputs.

## 7.2.6 Conclusion

Activation functions are a fundamental component of neural networks, enabling them to model non-linear relationships in data. The choice of activation function can significantly affect both the performance and convergence speed of the network. While traditional functions like sigmoid and tanh are still used in certain contexts, modern functions such as ReLU and its variants have become more popular due to their computational efficiency and ability to address common issues like the vanishing gradient problem.

**Table of Activation Functions (Part 1)**

Name	Function, $g(x)$	Derivative of $g$ , $g'(x)$	Range	Order of Continuity
Identity	$x$	1	$(-\infty, \infty)$	$C^\infty$
Binary Step	$\begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$	0	$\{0, 1\}$	$C^{-1}$
Logistic	$\sigma(x) = \frac{1}{1 + e^{-x}}$	$\sigma(x)(1 - \sigma(x))$	$(0, 1)$	$C^\infty$
Tanh	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - \tanh^2(x)$	$(-1, 1)$	$C^\infty$
ReLU	$g(x) = \max(0, x)$	$\begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$	$[0, \infty)$	$C^0$

Table 7.1: Table of Activation Functions (Part 1)

**Table of Activation Functions (Part 2)**

Name	Function, $g(x)$	Derivative of $g$ , $g'(x)$	Range	Order of Continuity
GELU	$\frac{1}{2}x \left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$	$\Phi(x) + x\phi(x)$	$(-0.17, \infty)$	$C^\infty$
Softplus	$\ln(1 + e^x)$	$\frac{1}{1 + e^{-x}}$	$(0, \infty)$	$C^\infty$
Leaky ReLU	$\begin{cases} 0.01x, & x < 0 \\ x, & x \geq 0 \end{cases}$	$\begin{cases} 0.01, & x < 0 \\ 1, & x \geq 0 \end{cases}$	$(-\infty, \infty)$	$C^0$
Swish	$\frac{x}{1 + e^{-x}}$	$\sigma(x) + x\sigma'(x)$	$(-0.278, \infty)$	$C^\infty$
Sinusoid	$\sin(x)$	$\cos(x)$	$[-1, 1]$	$C^\infty$

Table 7.2: Table of Activation Functions (Part 2)

## 7.3 Artificial Neural Networks

### 7.3.1 Introduction to Artificial Neural Networks

Neural networks are **foundational to machine learning**, inspired by biological neural systems. The network structure includes nodes (neurons) connected by weighted edges, allowing modeling of complex, non-linear data relationships.

### 7.3.2 Neurons

In the context of machine learning and neural networks, neurons are fundamental units that mimic the behavior of biological neurons in the human brain. A biological neuron receives signals from other neurons, processes them, and passes on a signal to other neurons, depending on the strength of the input and its internal state. Similarly, artificial neurons receive inputs, apply a function to process them, and generate an output. The concept of an artificial neuron was inspired by the biological process, and it serves as the building block of neural networks, particularly in deep learning architectures.

#### 7.3.2.1 Structure of a Neuron

In artificial neural networks, a neuron receives a set of inputs, typically represented as a vector  $\mathbf{x} = [x_1, x_2, \dots, x_n]$ , each with an associated weight  $w_i$ . The inputs are weighted and summed together to form a single value:

$$z = \sum_{i=1}^n w_i x_i + b$$

where  $b$  represents the bias term, which allows the neuron to adjust its output independently of the inputs. This sum  $z$  is then passed through an activation function, which introduces non-linearity to the model, enabling it to learn more complex patterns. Common activation functions include sigmoid, hyperbolic tangent ( $\tanh$ ), and rectified linear unit (ReLU).

#### 7.3.2.2 Biological Inspiration

In a biological system, neurons are specialized cells responsible for transmitting electrical and chemical signals. Each neuron has dendrites that receive signals from other neurons, a cell body that processes these signals, and an axon that transmits the output to other neurons or muscles. Neurons communicate with each other at synapses, where the strength of connections (synaptic weights) can change through learning mechanisms, such as synaptic plasticity. Similarly, in neural networks, weights adjust during training to minimize the error between predicted and actual outputs, analogous to how biological systems learn.

### 7.3.2.3 Artificial Neurons vs. Biological Neurons

While the operation of artificial neurons is inspired by biological neurons, they are much simpler in comparison. A biological neuron can involve complex processes, such as signal integration, synaptic learning, and neurotransmitter release, which are not captured by their artificial counterparts. However, the core principle remains similar: both systems take inputs, process them, and produce outputs.

### 7.3.2.4 Learning and Adaptation

Neurons in artificial neural networks learn by adjusting their weights and biases based on the error in the output of the network. This is done through a process called backpropagation, where the error is propagated backward through the network, and the weights are updated using an optimization algorithm such as gradient descent. This learning process is iterative, allowing the network to progressively improve its performance over time. In contrast, biological neurons adjust their synaptic strengths based on experience, a process which is influenced by biological mechanisms like Hebbian learning and synaptic plasticity.

In conclusion, artificial neurons, inspired by biological neurons, are essential components of neural networks, enabling them to process information and learn from data. While they operate on much simpler principles, their ability to adjust and improve over time makes them powerful tools for a wide range of machine learning tasks.

## 7.3.3 Basic Structure of Neural Networks

A neural network typically includes:

- **Input Layer:** Receives input data, with each node representing a feature.
- **Hidden Layers:** Intermediate layers process inputs using weighted connections. Networks can have multiple hidden layers, refining data.
- **Output Layer:** The final layer produces output predictions or classifications.

An activation function determines the range of values of activation of an artificial neuron. This is applied to the sum of the weighted input data of the neuron. An activation function is characterized by the property of non-linearity. Without the application of an activation function, the only operations in computing the output of a multilayer perceptron would be the linear products between the weights and the input values.

Linear operations performed in succession can be considered as a single linear operation. On the other hand, the application of the non-linear activation function leads to a non-linearity of the artificial neural network and thus to a non-linearity of the function that approximates the neural network. According to the approximation theorem, a multilayer perceptron with only one hidden layer using a nonlinear activation function is a universal function approximator.

Each neuron applies an *activation function* to determine if it should "fire" or transmit information. Common activation functions include:

- Sigmoid function:  $f(x) = \frac{1}{1 + e^{-x}}$
- tanh:  $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- ReLU (Rectified Linear Unit):  $f(x) = \max(0, x)$

**Note:**

- tanh is more popular than sigmoid function thanks to its zero-centered attribute. However, both functions *kill* gradient.
- In practice, ReLU has been shown to accelerate the convergence of the gradient descent toward the global minimum of the loss function compared to other activation functions. This is due to its linear, non-saturating property.
- While other activation functions (tanh and sigmoid) require very computationally expensive operations such as exponents, etc., ReLU, on the other hand, can be implemented simply by thresholding a value vector at zero.
- Unfortunately, there is also a problem with the ReLU activation function. Since the outputs of this function are zero for input values below zero, the neurons of the network can become very fragile and even "die" during training. What is meant by this? It may (need not, but can) happen that when the weights are updated, the weights are adjusted in a way that for certain neurons of a hidden layer the inputs  $x < 0$  are always below zero. This means that the values  $f(x)$  of these neurons (with  $f$  as the ReLU function) are always zero ( $f(x) = 0$ ) and thus make no contribution to the training process. This means that the gradient flowing through these ReLU neurons is also zero from that point on. We say that the neurons are *dead*. For example, it is very common to observe that 20-50% of the entire neural network that used the ReLU function is *dead*. In other words, these neurons are never activated in the entire dataset used in training.

## Logistic regression

Logistic regression is the simplest neural network model with only an input layer and an output layer.

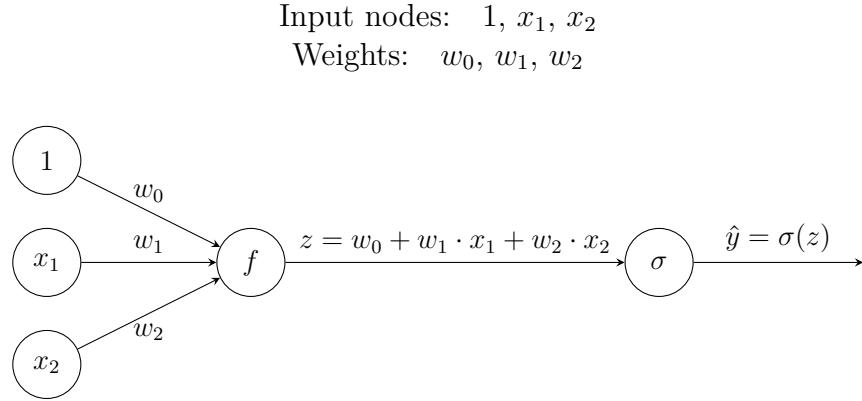
The model of logistic regression is represented as:

$$\hat{y} = \sigma(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2)$$

It consists of two steps:

- **Linear summation:** Compute  $z$ :  $z = 1 \cdot w_0 + x_1 \cdot w_1 + x_2 \cdot w_2$
- **Apply sigmoid function:**  $\hat{y} = \sigma(z)$

The following diagram illustrates the structure of logistic regression:



The first layer is the input layer, the intermediate layers are called hidden layers, and the last layer is the output layer. The circles in the diagram are called nodes.

Each model always has:

- 1 input layer,
- 1 output layer,
- Optional hidden layers in between.

The total number of layers in the model is defined as the number of layers minus 1 (excluding the input layer).

### Node Connections

Each node in the hidden layers and output layer:

- Is connected to every node in the previous layer with its own set of weights  $w$ .
- Has its own bias  $b$ .
- Performs two steps:
  1. Compute the linear summation.
  2. Apply the activation function.

### Notations

- The number of nodes in the  $i$ -th hidden layer is denoted as  $l^{(i)}$ .

- The weight matrix  $W^{(k)}$  between layer  $(k - 1)$  and layer  $k$  has dimensions  $l^{(k-1)} \times l^{(k)}$ .
- Each weight  $w_{ij}^{(k)}$  in  $W^{(k)}$  represents the connection from node  $i$  in layer  $(k - 1)$  to node  $j$  in layer  $k$ .

### Node Computations in Layer $k$

The bias vector for layer  $k$  is denoted as  $\mathbf{b}^{[k]}$  with size  $\ell^{[k]} \times 1$ , where  $b_i^{[k]}$  represents the bias for the  $i$ -th node in layer  $k$ .

Each node in layer  $\ell^{th}$  with bias  $b_i^{[\ell]}$  performs the following two steps:

- **Step 1: Linear summation**

$$z_i^{[\ell]} = \sum_{j=1}^{\ell^{[\ell-1]}} a_j^{[\ell-1]} \cdot w_{ji}^{[\ell]} + b_i^{[\ell]}$$

This computes the weighted sum of all nodes in the previous layer multiplied by their corresponding weights and adds the bias  $b_i^{[\ell]}$ .

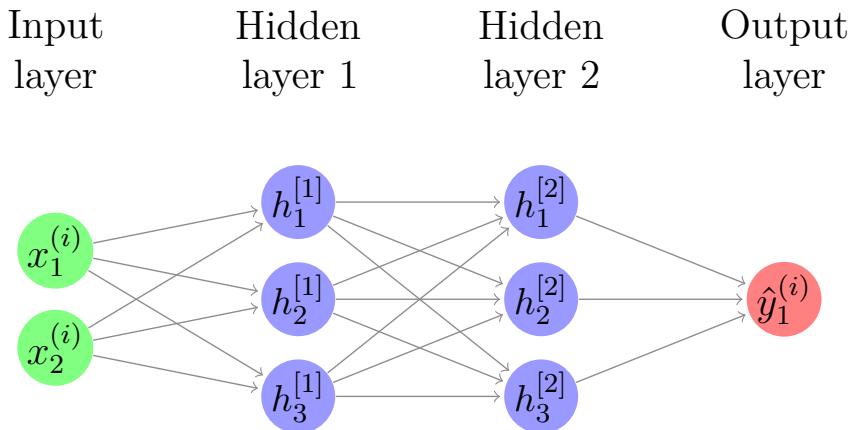
- **Step 2: Activation function**

$$a_i^{[\ell]} = \sigma(z_i^{[\ell]})$$

This applies the activation function (e.g., sigmoid or ReLU) to the linear summation result.

### Vector Representations for Layer $k$

- $\mathbf{z}^{[k]}$ : A vector of size  $l^{[k]} \times 1$  representing the linear summation values of all nodes in layer  $k$ .
- $\mathbf{a}^{[k]}$ : A vector of size  $l^{[k]} \times 1$  representing the values of all nodes in layer  $k$  after applying the activation function.



This neural network consists of 3 layers:

- Input layer with 2 nodes ( $l^{[0]} = 2$ ),
- Hidden layer 1 with 3 nodes ( $l^{[1]} = 3$ ),
- Hidden layer 2 with 3 nodes ( $l^{[2]} = 3$ ),
- Output layer with 1 node ( $l^{[3]} = 1$ ).

Since each node in the hidden layers and output layer has a bias term, an additional bias node is included in the input layer and hidden layers for the computation (but it is not counted in the total number of nodes for each layer).

### Computation for Node Values

For the second node in layer 1:

$$\begin{aligned} z_2^{[1]} &= x_1 \cdot w_{12}^{[1]} + x_2 \cdot w_{22}^{[1]} + b_2^{[1]} \\ a_2^{[1]} &= \sigma(z_2^{[1]}) \end{aligned}$$

For the third node in layer 2:

$$\begin{aligned} z_3^{[2]} &= a_1^{[1]} \cdot w_{13}^{[2]} + a_2^{[1]} \cdot w_{23}^{[2]} + a_3^{[1]} \cdot w_{33}^{[2]} + b_3^{[2]} \\ a_3^{[2]} &= \sigma(z_3^{[2]}) \end{aligned}$$

#### 7.3.4 Feedforward

To ensure consistent notation, the input layer is denoted as  $a^{[0]} = x$ , where  $a^{[0]}$  is of size  $2 \times 1$ .

The feedforward computation for layer 1 is as follows:

$$\mathbf{z}^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix} = \begin{bmatrix} a_1^{[0]} \cdot w_{11}^{[1]} + a_2^{[0]} \cdot w_{21}^{[1]} + b_1^{[1]} \\ a_1^{[0]} \cdot w_{12}^{[1]} + a_2^{[0]} \cdot w_{22}^{[1]} + b_2^{[1]} \\ a_1^{[0]} \cdot w_{13}^{[1]} + a_2^{[0]} \cdot w_{23}^{[1]} + b_3^{[1]} \end{bmatrix} = (W^{[1]})^T \cdot \mathbf{a}^{[0]} + \mathbf{b}^{[1]}$$

The activation values for layer 1 are computed as:

$$\mathbf{a}^{[1]} = \sigma(\mathbf{z}^{[1]})$$

Similarly, for the feedforward process:

$$\begin{aligned} z^{[2]} &= (W^{[2]})^T \cdot \mathbf{a}^{[1]} + b^{[2]}, \quad \mathbf{a}^{[2]} = \sigma(z^{[2]}) \\ z^{[3]} &= (W^{[3]})^T \cdot \mathbf{a}^{[2]} + b^{[3]}, \quad \hat{y} = \mathbf{a}^{[3]} = \sigma(z^{[3]}) \end{aligned}$$

The following diagram illustrates the feedforward process step-by-step:

$$a^{[0]} \longrightarrow z^{[1]} \longrightarrow a^{[1]} \longrightarrow z^{[2]} \longrightarrow a^{[2]} \longrightarrow z^{[3]} \longrightarrow a^{[3]} = \hat{y}$$

When working with datasets, we often compute predictions for multiple data points simultaneously. Let  $X$  represent a matrix of size  $n \times d$ , where:

- $n$  is the number of data points,
- $d$  is the number of features in each data point.

The value  $x_j^{(i)}$  represents the  $j$ -th feature of the  $i$ -th data point in the dataset. For example, given the dataset:

Salary	Working hours
10	1
5	2
7	0.15
6	1.8

Here:

$$n = 4, \quad d = 2, \quad x_1^{(1)} = 10, \quad x_2^{(1)} = 1, \quad x_1^{(3)} = 6, \quad x_2^{(3)} = 2$$

The input matrix  $X$  is written as:

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(n)} & x_2^{(n)} & \cdots & x_d^{(n)} \end{bmatrix} = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(n)})^T \end{bmatrix}$$

Since  $x^{(1)}$  is a vector of size  $d \times 1$ , each data point is represented as a row in  $X$ , requiring a transpose. Thus:

$$x^{(1)} \rightarrow (x^{(1)})^T$$

## Matrix Representations in Feedforward

Define:

- $Z^{(i)}$ : A matrix of size  $N \times l^{(i)}$ , where  $z_j^{(i)[k]}$  is the value of the  $j$ -th node in layer  $i$  for the  $k$ -th data point after the linear summation.
- $A^{(i)}$ : A matrix of size  $N \times l^{(i)}$ , where  $a_j^{(i)[k]}$  is the value of the  $j$ -th node in layer  $i$  for the  $k$ -th data point after applying the activation function.

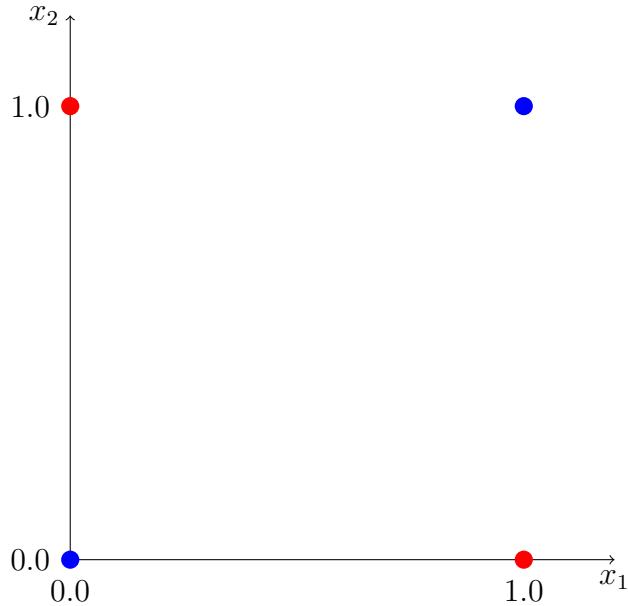
## 7.3.5 Backpropagation

### 7.3.5.1 Perceptron Limitation: The XOR problem

The XOR ( $\oplus$ ) operation for 2-bit input returns 1 if exactly one of the two bits is 1, and returns 0 in all other cases. The truth table is as follows:

$A$	$B$	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

When the XOR problem is modeled using logistic regression, the resulting plot is shown below:



Clearly, a single straight line cannot separate the data points into two regions. This demonstrates that applying gradient descent to the XOR problem—regardless of how many steps you take or how you tune the learning rate—cannot produce the desired outcome. Perceptrons (Traditional Linear functions) is insufficient for solving the XOR problem. We need a new solution to address this challenge.

$$A \oplus B = (\neg(A \wedge B)) \wedge (A \vee B)$$

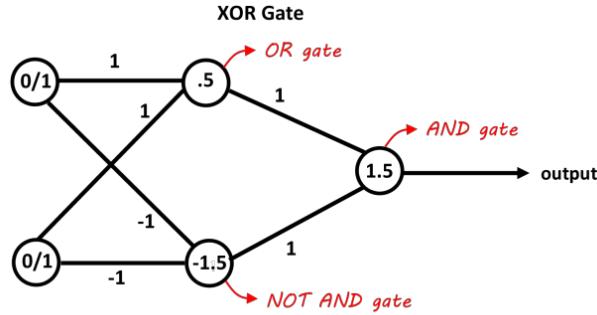


Figure 7.1: XOR as a neural network

**Input layer:** the two inputs 0/1 represent the binary inputs  $A$  and  $B$ .

**Hidden layer:**

- **Neuron 1 (OR Gate):** Weights = 1 for both inputs  $A$  and  $B$ , Bias: 0.5,
- **Neuron 2 (NOT AND Gate):** Weights:  $-1$  for both inputs  $A$  and  $B$ , Bias:  $-1.5$ , ensuring the neuron models NOT AND behavior.

**Output Layer:** Weights: 1, Bias: 1.5.

### 7.3.5.2 The Backpropagation Process

Take the example from Figure 7.3.3 to elaborate.

For each data point  $(x^{[i]}, y_i)$ , the loss function is defined as:

$$L = -\left( y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i) \right)$$

where:

$$\hat{y}_i = a_1^{[2]} \cdot w_{11}^{[2]} + a_2^{[1]} \cdot w_{21}^{[2]} + b_1^{[2]}$$

is the predicted value by the model, and  $y_i$  is the true label of the data point.

The derivative of  $L$  with respect to  $\hat{y}_i$  is:

$$\begin{aligned} \frac{\partial L}{\partial \hat{y}_i} &= \frac{\partial}{\partial \hat{y}_i} \left( -y_i \cdot \log(\hat{y}_i) - (1 - y_i) \cdot \log(1 - \hat{y}_i) \right) \\ &= -\left( \frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i} \right) \end{aligned}$$

For the entire dataset, the loss function  $J$  is computed as:

$$J = -\frac{1}{N} \sum_{i=1}^N \left( y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i) \right)$$

In Backpropagation, we reverse the process of Feed-forwarding by calculating these values:

$$\frac{\partial J}{\partial W^{[1]}} \leftarrow \frac{\partial J}{\partial b^{[1]}} \leftarrow \frac{\partial J}{\partial A^{[1]}} \leftarrow \frac{\partial J}{\partial W^{[2]}} \leftarrow \frac{\partial J}{\partial b^{[2]}} \leftarrow \frac{\partial J}{\partial A^{[2]}} \leftarrow \frac{\partial J}{\partial W^{[3]}} \leftarrow \frac{\partial J}{\partial b^{[3]}} \leftarrow \frac{\partial J}{\partial \hat{Y}}$$

1. **Step 1:** Compute  $\frac{\partial J}{\partial \hat{Y}}$ , where  $\hat{Y} = A^{[3]}$ .

2. **Step 2:** Compute:

$$\begin{aligned} \frac{\partial J}{\partial W^{[3]}} &= (A^{[2]})^T \cdot \left( \frac{\partial J}{\partial \hat{Y}} \odot \frac{\partial A^{[3]}}{\partial Z^{[3]}} \right) \\ \frac{\partial J}{\partial b^{[3]}} &= \left( \frac{\partial J}{\partial \hat{Y}} \odot \frac{\partial A^{[3]}}{\partial Z^{[3]}} \right)^T \cdot \mathbf{1} \end{aligned}$$

Compute:

$$\frac{\partial J}{\partial A^{[2]}} = \left( \frac{\partial J}{\partial \hat{Y}} \odot \frac{\partial A^{[3]}}{\partial Z^{[3]}} \right) \cdot (W^{[3]})^T$$

3. **Step 3:** Compute:

$$\begin{aligned} \frac{\partial J}{\partial W^{[2]}} &= (A^{[1]})^T \cdot \left( \frac{\partial J}{\partial A^{[2]}} \odot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \right) \\ \frac{\partial J}{\partial b^{[2]}} &= \left( \frac{\partial J}{\partial A^{[2]}} \odot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \right)^T \cdot \mathbf{1} \end{aligned}$$

Compute:

$$\frac{\partial J}{\partial A^{[1]}} = \left( \frac{\partial J}{\partial A^{[2]}} \odot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \right) \cdot (W^{[2]})^T$$

4. **Step 4:** Compute:

$$\begin{aligned} \frac{\partial J}{\partial W^{[1]}} &= (A^{[0]})^T \cdot \left( \frac{\partial J}{\partial A^{[1]}} \odot \frac{\partial A^{[1]}}{\partial Z^{[1]}} \right) \\ \frac{\partial J}{\partial b^{[1]}} &= \left( \frac{\partial J}{\partial A^{[1]}} \odot \frac{\partial A^{[1]}}{\partial Z^{[1]}} \right)^T \cdot \mathbf{1} \end{aligned}$$

Here  $A^{[0]} = X$ .

### 7.3.6 Types of Neural Network Architectures

Neural networks are categorized based on structure and data flow:

- **Feedforward Neural Networks (FNN):** Data moves in one direction from input to output. Used for general tasks like classification and regression.
- **Convolutional Neural Networks (CNN):** Used for image processing with layers for feature extraction and dimensionality reduction.
- **Recurrent Neural Networks (RNN):** Includes feedback loops for sequential data, ideal for tasks like language modeling.
- **Transformers:** Uses an attention mechanism, making it efficient for language processing tasks.
- **Autoencoders:** Used for unsupervised learning like dimensionality reduction and anomaly detection.

### 7.3.7 Training Neural Networks

Training a neural network optimizes connection weights through *backpropagation*:

1. **Forward Propagation:** Data moves from input to output layers.
2. **Loss Calculation:** Measures prediction error using functions such as Mean Squared Error or Cross-Entropy.
3. **Backpropagation:** Uses gradient descent to update weights and minimize error.

### Regularization Techniques

To enhance performance and prevent overfitting, techniques include:

- **Regularization (L1, L2):** Adds penalties to weights, discouraging complex models.
- **Dropout:** Temporarily removes neurons to reduce dependency on specific paths.
- **Batch Normalization:** Stabilizes learning by normalizing layer inputs.

### 7.3.8 Advanced Optimization Techniques

- **Stochastic Gradient Descent (SGD):** Uses mini-batches for faster, more stable updates.
- **Adam (Adaptive Moment Estimation):** Combines SGD with adaptive learning rates, popular for deep learning.

### 7.3.9 Applications of Neural Networks

Neural networks have numerous applications:

- **Image Processing:** CNNs are used in object detection and facial recognition.
- **Natural Language Processing:** RNNs and Transformers handle tasks like language translation and sentiment analysis.
- **Time Series and Financial Forecasting:** RNNs analyze sequential data such as stock prices.
- **Healthcare:** Neural networks aid in diagnosing diseases from medical data.
- **Generative Models:** Autoencoders and GANs generate realistic imgs and audio.

### 7.3.10 Future Directions and Challenges

Future directions include *explainable AI* for interpretability and *quantum neural networks* for merging quantum computing with deep learning. Challenges include high computational resources, large datasets, and ethical considerations.

**Note:** Will propose a low-level detail of Regularization, Normalization in next step.

## 7.4 Feed-Forward Neural Networks

Feed-forward neural networks are the most fundamental type of artificial neural network. In these networks, information moves in a single direction—from the input layer, through hidden layers (if any), and finally to the output layer. Each layer consists of interconnected nodes, also known as neurons, that are responsible for processing input data and passing information forward. Unlike recurrent neural networks, **feed-forward networks have no feedback loops, meaning outputs do not influence inputs, and neurons in one layer do not have connections back to neurons in previous layers.**

### 7.4.1 Structure of Feed-Forward Neural Networks

Feed-forward neural networks consist of three main types of layers:

- **Input Layer:** The first layer receives the raw input data. Each neuron in this layer represents a specific feature of the input, such as pixel values in an image or words in a text.
- **Hidden Layers:** Intermediate layers between the input and output layers. These layers extract and process features from the input data by applying weights and activation functions. Networks may have multiple hidden layers, with each layer allowing for more complex representations of the input data.
- **Output Layer:** The final layer produces the network's output, which could be a classification, regression result, or other prediction, depending on the task.

### 7.4.2 Mathematics of Feed-Forward Neural Networks

In feed-forward neural networks, each neuron in a layer calculates a weighted sum of the inputs it receives from the previous layer and applies an activation function to produce its output. Mathematically, the output of the  $j^{th}$  neuron in the  $\ell^{th}$  layer can be expressed as:

$$y_j^{[\ell]} = f \left( \sum_{i=1}^n w_{ij}^{[\ell-1]} y_i^{[\ell-1]} + b_j^{[\ell]} \right)$$

where:

- $y_i^{[\ell-1]}$  is the output of the  $i^{th}$  neuron in the previous layer,
- $w_{ij}^{[\ell-1]}$  is the weight connecting the  $i^{th}$  neuron in the  $(\ell - 1)^{th}$  layer to the  $j^{th}$  neuron in the  $\ell^{th}$  layer,
- $b_j^{[\ell]}$  is the bias term for the  $j^{th}$  neuron in the  $\ell^{th}$  layer,
- $f$  is the activation function, which introduces non-linearity to enable the network to learn complex mappings.

### 7.4.3 Activation Functions in Feed-Forward Neural Networks

Activation functions are **essential for introducing non-linearity** into neural networks, allowing them to model complex relationships in data. Common activation functions include:

- **Sigmoid Function:** Maps inputs to a range between 0 and 1, often used for binary classification tasks.
- **Hyperbolic Tangent ( $\tanh$ ):** Maps inputs to a range between -1 and 1, which can help in centering the output.
- **Rectified Linear Unit (ReLU):** The most widely used activation function, which outputs zero for negative inputs and the input itself for positive values, helping to mitigate the vanishing gradient problem.

### 7.4.4 Training Feed-Forward Neural Networks

Feed-forward neural networks are trained using supervised learning techniques, where labeled data is used to optimize the network's parameters. Training typically involves:

- **Forward Propagation:** Data is passed through the network to calculate the output.
- **Loss Calculation:** The error between the predicted output and the actual target is computed using a loss function.
- **Backpropagation:** The gradients of the loss with respect to the network's weights are calculated and used to update weights in order to minimize the error.
- **Optimization:** Optimizers such as stochastic gradient descent or Adam are employed to adjust weights and biases based on the gradients obtained in backpropagation.

### 7.4.5 Applications of Feed-Forward Neural Networks

Feed-forward neural networks are used across a wide range of applications:

- **Image Classification:** Used in systems that classify objects or scenes in images.
- **Natural Language Processing:** Used in text classification, sentiment analysis, and language modeling tasks.
- **Regression Tasks:** Applied to problems where the goal is to predict a continuous output, such as predicting housing prices or weather patterns.

#### 7.4.6 Limitations of Feed-Forward Neural Networks

While feed-forward neural networks are powerful for many applications, they have limitations:

- **Lack of Memory:** They cannot retain information about previous inputs, making them unsuitable for tasks requiring sequential data or temporal dependencies.
- **Computationally Intensive:** Training large networks can be computationally expensive and time-consuming.
- **Susceptible to Overfitting:** Feed-forward neural networks can easily overfit on small datasets, making regularization techniques essential.

#### 7.4.7 Summary

Feed-forward neural networks are foundational models in machine learning, offering a structured approach to transforming input data into useful outputs through a series of weighted connections and activation functions. Their simplicity, combined with their capability for modeling complex, non-linear relationships, makes them a cornerstone of many machine learning applications. Despite their limitations, feed-forward neural networks remain widely used and form the basis for more advanced architectures.

## 7.5 Stochastic Neural Networks

Stochastic neural networks are neural networks that incorporate randomness into their structure or function, which allows them to capture uncertainties in data and model complex probabilistic relationships. **Unlike traditional deterministic neural networks, where a given input always results in a specific output, stochastic neural networks can produce varied outputs for the same input due to their probabilistic components.** This property makes stochastic neural networks suitable for tasks that require flexibility in decision-making or need to account for uncertainty in predictions.

### 7.5.1 Introduction to Stochasticity in Neural Networks

In stochastic neural networks, randomness can be introduced at various levels:

- **Randomized Weights:** Weights in the network can follow a distribution instead of being fixed values, making the network's behavior probabilistic.
- **Stochastic Activation Functions:** Some networks use activation functions that incorporate randomness, leading to variable outputs even with identical inputs.
- **Randomized Layers and Dropout:** Techniques like dropout randomly deactivate a subset of neurons during each training iteration, helping the network learn a more generalized model.

### 7.5.2 Advantages of Stochastic Neural Networks

The stochastic nature of these networks provides several key benefits:

- **Improved Generalization:** By learning to predict distributions rather than fixed outputs, stochastic neural networks tend to generalize better, especially in data-limited settings.
- **Robustness to Noise:** Stochastic components make the networks less sensitive to noise in input data, resulting in more stable predictions.
- **Uncertainty Estimation:** Stochastic neural networks inherently provide uncertainty estimates in their predictions, which can be crucial for applications where decision confidence is needed.

### 7.5.3 Training Methods for Stochastic Neural Networks

Training stochastic neural networks involves methods that account for their probabilistic nature. Common techniques include:

- **Bayesian Training Approaches:** Using Bayesian inference, these networks can estimate posterior distributions of weights based on observed data, leading to networks that represent weight uncertainty.
- **Variational Inference:** This approach approximates the posterior distribution of weights by optimizing a family of distributions, enabling efficient training with stochastic gradient descent.
- **Monte Carlo Dropout:** A practical approximation of Bayesian inference where dropout is applied at inference time, resulting in predictions that are averaged over multiple passes through the network.

#### 7.5.4 Applications of Stochastic Neural Networks

Stochastic neural networks are highly valued in fields where predictive uncertainty and robustness are essential. Applications include:

- **Medical Diagnosis:** Stochastic neural networks help quantify uncertainty in predictions, which is important for high-stakes decisions.
- **Autonomous Systems:** In areas such as self-driving vehicles, uncertainty in obstacle detection or navigation can enhance safety.
- **Financial Modeling:** These networks can aid in decision-making under uncertainty, which is crucial for managing financial risks.

#### 7.5.5 Challenges and Future Directions

While stochastic neural networks offer significant advantages, they also come with challenges:

- **Computational Complexity:** The need for sampling and probabilistic inference can make training and inference more computationally demanding.
- **Interpretability:** Interpreting probabilistic models can be more challenging, as predictions are distributions rather than fixed values.

Future research aims to improve the efficiency of training methods, scalability to larger datasets, and integration of interpretability techniques to make stochastic neural networks more accessible and useful in practical applications.

Stochastic neural networks bring a probabilistic approach to deep learning, making them a powerful tool for scenarios that require the modeling of uncertainty. By learning distributions over weights and incorporating randomness at various levels, these networks provide robust predictions with uncertainty quantification, which is valuable in fields ranging from medicine to finance. Continued advancements in training techniques and computational efficiency will further enhance their applicability and effectiveness in real-world problems.

### 7.5.6 Bayesian Neural Networks

Bayesian neural networks (BNNs) represent a prominent category of stochastic neural networks trained using a Bayesian framework. Rather than learning fixed weights, BNNs learn probability distributions over weights, allowing them to capture model uncertainty. This makes BNNs suitable for applications requiring accurate uncertainty estimates, such as medical diagnosis or autonomous driving.

#### Bayesian Inference and Weight Distributions

In BNNs, each weight is associated with a probability distribution, reflecting the uncertainty over its values. Training a BNN involves calculating the posterior distribution of the weights given the data, which is often achieved through **Bayesian inference** techniques. Unlike traditional neural networks, where weights are point estimates, BNNs seek the posterior distribution  $p(\mathbf{W}|\mathbf{D})$  where

$$p(\mathbf{W}|\mathbf{D}) = \frac{p(\mathbf{D}|\mathbf{W}) \cdot p(\mathbf{W})}{p(\mathbf{D})}.$$

$p(\mathbf{D}|\mathbf{W})$  is the **likelihood** of the data given the weights,

$p(\mathbf{W})$  is the **prior distribution** over weights,

$p(\mathbf{D})$  is the **marginal likelihood** of the data.

Exact inference in BNNs is computationally challenging, so approximate methods like **Variational Inference** or **Markov Chain Monte Carlo (MCMC)** are commonly used.

### 7.5.7 Benefits of Bayesian Neural Networks

BNNs offer key advantages due to their probabilistic nature:

- **Uncertainty Quantification:** BNNs provide uncertainty estimates alongside predictions, which is beneficial in high-stakes scenarios, enabling more cautious decision-making.
- **Improved Generalization:** Through the use of weight distributions, BNNs often generalize better, especially with limited data.
- **Regularization Effect:** The Bayesian prior acts as a regularizer, discouraging overfitting by favoring simpler models.

### 7.5.8 Challenges and Limitations of Bayesian Neural Networks

Despite their benefits, BNNs have limitations:

- **Computational Complexity:** Training a BNN requires complex computations for approximating the posterior distribution, which can be time-consuming.

- **Scalability:** Due to the computational demands of Bayesian inference, BNNs can be challenging to scale to large datasets or architectures.

BNNs continue to gain attention as methods improve, particularly with advancements in variational inference and sampling techniques that make Bayesian inference in neural networks more tractable.

### 7.5.9 Applications of Bayesian Neural Networks

BNNs are particularly valuable in applications where uncertainty is crucial, such as:

- **Medical Diagnostics:** BNNs provide confidence estimates, aiding in scenarios where predictive uncertainty can impact treatment decisions.
- **Autonomous Vehicles:** BNNs help in navigation and object detection, where uncertainty estimates improve safety.
- **Financial Modeling:** Probabilistic predictions are beneficial in forecasting stock prices and risk assessment.

Thus, Bayesian neural networks extend neural network capabilities by integrating probabilistic reasoning, allowing models to make well-calibrated predictions and enhance trustworthiness in AI applications.

## 7.6 Convolutional Neural Networks

Convolutional Neural Networks are a specialized type of artificial neural network designed for processing structured grid data, such as images. Originally inspired by the visual cortex in animals, Convolutional Neural Networks excel in handling spatial and temporal dependencies in data, making them particularly effective for image processing and computer vision tasks. Unlike traditional neural networks, Convolutional Neural Networks use convolutional layers that apply filters to detect features in local regions of the input.

### 7.6.1 Architecture of Convolutional Neural Networks

The architecture of a Convolutional Neural Network typically includes the following key layers:

- **Convolutional Layer:** This layer applies a set of learnable filters (also known as kernels) across the input image. *Each filter detects specific features*, such as edges or textures, in the local regions of the input.
- **Pooling Layer:** Often following convolutional layers, pooling layers *reduce the spatial dimensions* of the data, which decreases computational requirements and reduces overfitting. Common pooling methods include max pooling and average pooling.
- **Fully Connected Layer:** Located near the end of the network, these layers *interpret the high-level features extracted by previous layers* and generate predictions.
- **Activation Functions:** Activation functions, such as Rectified Linear Unit, introduce non-linearity into the model, enabling it to learn complex patterns. Convolutional Neural Networks typically use Rectified Linear Unit as the activation function.

### 7.6.2 Convolutional Operation

*Link to the animation of convolutional arithmetics here*

To make it easier to visualize, I will use an example on a grayscale image, which means the image is represented as a matrix  $A$  of size  $m \times n$ .

We define a kernel as a square matrix of size  $k \times k$ , where  $k$  is an odd number.  $k$  can be  $1, 3, 5, 7, 9, \dots$ . For example, a kernel of size  $3 \times 3$ :

$$W = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The notation for the convolution operation is  $(\otimes)$ , and we denote  $Y = X \otimes W$ .

For each element  $x_{ij}$  in matrix  $X$ , we extract a sub-matrix of size equal to the size of the kernel  $W$  with element  $x_{ij}$  at the center (this is why the kernel size is usually odd), and we call this matrix  $A$ . Then, we calculate the sum of the element-wise product of matrix  $A$  and matrix  $W$ , and write the result into matrix  $Y$ .

$$\begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 1 & 0 \\ \hline \end{array} \otimes \begin{array}{|c|c|c|} \hline 1 & 0 & 1 \\ \hline 0 & 1 & 0 \\ \hline 1 & 0 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline \textcolor{blue}{\boxed{}} & & \\ \hline & & \\ \hline & & \\ \hline \end{array}$$

**X**                                   **W**                                   **Y**

For example, to calculate  $y_{11}$ , we use the centering element  $x_{22}$  and form a  $3 \times 3$  submatrix, then perform an element-wise operation:

$$\begin{aligned} y_{11} &= \Sigma(\mathbf{A} \otimes \mathbf{W}) \\ &= x_{11}w_{11} + x_{12}w_{12} + x_{13}w_{13} + x_{21}w_{21} + x_{22}w_{22} + x_{23}w_{23} \\ &\quad + x_{31}w_{31} + x_{32}w_{32} + x_{33}w_{33} \\ &= 4. \end{aligned}$$

The rest is calculated accordingly. For boundary squares like  $x_{11}$ , we ignore them.

0	1	1	1	0
0	1	0	1	0
0	0	1	1	1
0	0	1	0	0
0	1	0	1	0

The matrix  $\mathbf{Y}$  has size of  $(m - k + 1)(n - k + 1)$  which is smaller than  $\mathbf{X}$ .

### 7.6.3 Padding

As described above, every time a convolution operation is performed, the size of matrix  $Y$  is always smaller than  $X$ . Since we typically use small kernels, for any given convolution we

might only lose a few pixels but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. We want matrix  $Y$  to have the **same size** as matrix  $X$ . A good thinking is to find a solution for elements at the border. Adding zeroes to the border of matrix  $X$  seems like a good idea.

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0

**Figure:** Matrix  $X$  after adding 0-padding on the border.

Clearly, the issue of finding matrix  $A$  for element  $x_{11}$  has now been resolved, and matrix  $Y$  will have the same size as the original matrix  $X$ .

This operation is called convolution with padding = 1. Padding =  $k$  means adding  $k$ -vector of 0 values to each side (top, bottom, left, right) of the matrix.

Generally, for an input matrix of size  $m \times n$  and a filter of size  $p \times q$ , with padding  $P$ , the output dimensions  $x$  and  $y$  of  $\mathbf{W}$  are calculated as:

- $x = m - p + 2P + 1$
- $y = n - q + 2P + 1$

#### 7.6.4 Stride

In Convolutional Neural Networks, stride refers to the number of steps the convolutional filter moves across the input matrix. A stride determines how much overlap exists between consecutive positions of the filter, impacting the output size:

- A **stride of 1** means the filter moves one step at a time, producing maximum overlap. This is used to preserve spatial dimensions, when feature extraction is the priority.
- A **stride of 2** results in downsampling, as the filter skips steps between positions, reducing the output size.
- **Stride of 3 or more:** Rarely used, as it can lose important spatial details.

## Examples

**Input Matrix:**

$$\mathbf{X} = \begin{bmatrix} 4 & 9 & 2 & 5 & 8 & 3 \\ 2 & 4 & 0 & 4 & 0 & 3 \\ 2 & 5 & 4 & 5 & 4 & 5 \\ 5 & 5 & 4 & 7 & 8 & 2 \\ 5 & 7 & 9 & 2 & 1 & 9 \\ 5 & 5 & 8 & 3 & 8 & 4 \end{bmatrix}$$

**Filter:**

$$\mathbf{W} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

**Stride = 1:** The filter moves one step at a time, fully covering the input matrix (but not the elements at the boundary).

Example:  $y_{12} = \sum_{i=1}^3 \sum_{j=1}^3 x_{i(j+1)} w_{ij} = 9 \cdot 1 + 2 \cdot 0 + 5 \cdot (-1) + 4 \cdot 1 + 0 \cdot 0 + 4 \cdot (-1) + 5 \cdot 1 + 4 \cdot 0 + 5 \cdot (-1) = 4$ .

$$\text{Output Matrix: } \mathbf{Y}_{\text{stride}=1} = \begin{bmatrix} 2 & 4 & -6 & 3 \\ 1 & -2 & -4 & 6 \\ -5 & 3 & 4 & -2 \\ -6 & 5 & 4 & -3 \end{bmatrix}$$

**Stride = 2:** The filter moves two steps at a time, skipping intermediate elements.

$$\text{Output Matrix: } \mathbf{Y}_{\text{stride}=2} = \begin{bmatrix} 2 & 4 \\ 1 & -2 \end{bmatrix}$$

**Padding = 1 (Zero padding), Stride = 2:**

$$\text{Output Matrix: } \mathbf{Y}_{\text{padding}=1, \text{stride}=2} = \begin{bmatrix} -13 & 4 & 3 \\ -14 & -2 & -6 \\ -17 & 5 & -3 \end{bmatrix}$$

Generally, for an input matrix of size  $m \times n$  and a filter of size  $p \times q$ , with padding  $P$  and stride  $s$ , the output dimensions  $x$  and  $y$  of  $\mathbf{W}$  are calculated as:

- $x = \frac{m - p + 2P}{s} + 1,$
- $y = \frac{n - q + 2P}{s} + 1$

The convolutional operation is the core of Convolutional Neural Networks, where filters are applied to input data to extract features. The output of this operation is called a **feature**

**map or activation map.** Mathematically, the convolution operation for an input  $X$  and  $k$  filters  $\mathbf{W}$  with a bias  $b$  can be expressed as:

$$\mathbf{Y}(\mathbf{i}, \mathbf{j}, \mathbf{k}) = f \left( \sum_{d=0}^{D-1} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \mathbf{X}(s\mathbf{i} + \mathbf{m} - \mathbf{p}, s\mathbf{j} + \mathbf{n} - \mathbf{p}, d) \cdot \mathbf{W}_k(\mathbf{m}, \mathbf{n}, d) + b_k \right)$$

where:

- $i, j$ : Row and column indices of the output feature map,
- $k$ : Index of the filter producing the  $k$ -th output channel,
- $m, n$ : Row and column indices of the filter/kernel,
- $s$ : The stride, which determines the step size of the filter's movement across the input,
- $p$ : The padding size, which adds  $p$  rows/columns of zeros around the input tensor,
- $D$ : The depth of the input tensor (number of channels),
- $M, N$ : The size of the filter,
- $Y(i, j, k)$ : The value at position  $(i, j)$  in the  $k$ -th output feature map,
- $f$ : The activation function (e.g., ReLU, tanh) applied element-wise after convolution,
- $X(s\mathbf{i} + \mathbf{m} - \mathbf{p}, s\mathbf{j} + \mathbf{n} - \mathbf{p}, d)$ : The input value at position  $(s\mathbf{i} + \mathbf{m} - \mathbf{p}, s\mathbf{j} + \mathbf{n} - \mathbf{p})$  in the  $d$ -th channel, accounting for stride  $s$  and padding  $p$ ,
- $W_k(m, n, d)$ : The value of the  $k$ -th filter (kernel) at position  $(m, n)$  in the  $d$ -th channel,
- $b_k$ : The bias term for the  $k$ -th filter.

The **output** of the convolutional layer will be a **tensor** with the following dimensions:

$$\left( \frac{I - p + 2P}{S} + 1 \right) \times \left( \frac{J - q + 2P}{S} + 1 \right) \times K$$

**Total Parameters of the Convolutional Layer** The total number of parameters for the layer is calculated as:

- Each kernel has  $p \times q \times D$  weights and 1 bias term.
- Total parameters per kernel:

$$p \times q \times D + 1$$

- For  $K$  kernels, the total parameters are:

$$K \times (p \times q \times D + 1)$$

The output of the first convolutional layer will be the input for the next convolutional layer.

### 7.6.5 1D Convolution in NLP

For a sequence of word embeddings:

- Input tensor  $X$  of size  $L \times D$ , where  $L$  is the sequence length, and  $D$  is the embedding size.
- A kernel (filter)  $W$  of size  $F \times D$ , where  $F$  is the filter size (length).
- Stride  $S$  determines the step size of the kernel.
- Padding  $P$  adds zeros to the input sequence.
- Bias  $b_k$  is a scalar bias term for each filter.
- $K$  is the number of filters, producing  $K$  output channels.

The output of the 1D convolutional layer is given by:

$$Y(i, k) = f \left( \sum_{d=0}^{D-1} \sum_{m=0}^{F-1} X(si + m - p, d) \cdot W_k(m, d) + b_k \right)$$

Where:

- $Y(i, k)$  is the output at position  $i$  for the  $k$ -th filter.
- $X(si + m - p, d)$  is the input value at position  $si + m - p$  for the  $d$ -th embedding dimension.
- $W_k(m, d)$  is the weight of the  $k$ -th filter at position  $(m, d)$ .
- $f$  is the activation function (e.g., ReLU or tanh).
- The output tensor has dimensions:

$$\left( \frac{L - F + 2P}{S} + 1 \right) \times K$$

where  $K$  is the number of filters.

1D convolution is commonly used in NLP for tasks such as:

- Extracting n-gram features from sequences.
- Sentiment analysis and text classification.
- Feature extraction in character-level embeddings.

## 7.6.6 Popular kernels

Popular kernels have the size of  $F \times F$  ( $F$  is odd) since we can find a submatrix having the exact same size as the kernel while also having a *center* element. They are also square matrices.

### Edge Detection Kernel

Edge detection kernels highlight the edges or boundaries in an image. One common example is the Sobel kernel, used to detect horizontal and vertical edges.

Horizontal Sobel Kernel:

$$W_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Vertical Sobel Kernel:

$$W_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

### Box Blur Kernel

The box blur kernel smooths an image by averaging the pixel values within a local region. It is useful for preprocessing tasks e.g. reducing noise, or softening the image.

$$W_{\text{box blur}} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

### Identity Kernel

The identity kernel retains the original image without any modification. It is often used as a baseline for comparison.

$$W_{\text{identity}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

### Sharpening Kernel

The sharpening kernel enhances the edges and details in an image by amplifying the differences between neighboring pixel values. Useful in medical imaging and satellite imagery.

$$W_{\text{sharpen}} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

## 7.6.7 Pooling Layers

Pooling layers perform **down-sampling** operations that **reduce the spatial dimensions of the feature maps**, which helps in minimizing computational load and controlling overfitting.

The pooling layer is typically used between convolutional layers to reduce the spatial dimensions of the data while retaining the most important features. This dimensionality reduction helps decrease computational costs and avoids overfitting.

With pooling, the image size is reduced, allowing subsequent convolutional layers to learn features from a larger receptive field. For instance, an image of size  $224 \times 224$  can be reduced to  $112 \times 112$  using pooling with a filter of size  $2 \times 2$  and stride 2. Consequently, the convolutional layer processes smaller images but learns features across larger regions.

### Pooling Operation

Consider a pooling layer with:

- Pooling size  $K \times K$ ,
- Input of size  $H \times W \times D$  (Height, Width, Depth),
- Output calculated by applying the pooling operation (e.g., max or average) over each region of size  $K \times K$  in the input.

The stride  $S$  and padding  $P$  for pooling follow the same rules as those used in convolution operations.

### Example: Max Pooling

Below is an example of max pooling with pooling of size  $3 \times 3$ , Stride 1, and Padding 0:

Input matrix:

$$\text{Input} = \begin{bmatrix} 3 & 3 & 2 & 1 & 0 \\ 0 & 1 & 3 & 1 & 0 \\ 3 & 1 & 2 & 2 & 3 \\ 2 & 0 & 0 & 2 & 0 \\ 2 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Max pooling result:

$$\text{Output} = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 2 & 3 \end{bmatrix}$$

### Common Pooling Configurations

In most cases, pooling layers use a Pooling size  $2 \times 2$ , Stride 2, and Padding 0.

This configuration reduces the height and width of the data by half, while preserving the depth.

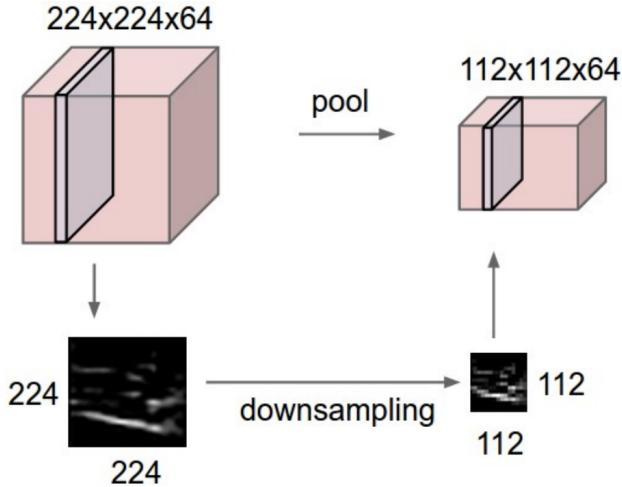


Figure 7.2: A pooling layer with pooling size 2x2

Two popular types of pooling are:

- **Max Pooling:** Selects the maximum value in each region of the feature map, preserving the most significant features.
- **Average Pooling:** Computes the average of all values in each region, which helps retain contextual information but can sometimes lose fine-grained details.

Pooling is often applied with a stride, which determines the step size with which the pooling window moves across the feature map.

### 7.6.8 Engineering with Convolutional Operations

Here is the Python code for carrying convolutional arithmetics with padding, stride, and pooling options:

Listing 7.1: Use Python for convolutional arithmetics

```

1 import numpy as np
2 import pandas as pd
3
4 def pad_matrix(X, pad_size):
5     """Apply zero-padding to the matrix."""
6     return np.pad(X, ((pad_size, pad_size), (pad_size, pad_size)), mode='constant', constant_values=0)
7
8 def convolution_2d(X, W, stride=1, padding="valid"):
9     """
10         Perform 2D convolution.
11         :param X: Input matrix
12         :param W: Filter/kernel
13         :param stride: Stride of the convolution

```

```

14     :param padding: "valid" for no padding or "same" for zero padding to
15         maintain output size
16     :return: Convolution result
17     """
18
19     if padding == "same":
20         pad_size = W.shape[0] // 2 # Assuming square filter
21         X = pad_matrix(X, pad_size)
22     elif padding != "valid":
23         raise ValueError("Padding must be 'valid' or 'same'")
24
25
26     # Calculate output dimensions
27     output_dim = ((X.shape[0] - W.shape[0]) // stride) + 1
28
29     # Initialize the output matrix
30     output = np.zeros((output_dim, output_dim))
31
32     # Perform the convolution
33     for i in range(output_dim):
34         for j in range(output_dim):
35             region = X[i*stride:i*stride+W.shape[0], j*stride:j*stride+W.
36                         shape[1]]
37             output[i, j] = np.sum(region * W)
38
39     return output
40
41
42 def pooling_2d(X, pool_size=2, pool_type="max", stride=2):
43     """
44     Perform 2D pooling.
45     :param X: Input matrix
46     :param pool_size: Size of the pooling window
47     :param pool_type: "max" for max pooling or "average" for average
48         pooling
49     :param stride: Stride of the pooling operation
50     :return: Pooled result
51     """
52
53     output_dim = ((X.shape[0] - pool_size) // stride) + 1
54     output = np.zeros((output_dim, output_dim))
55
56
57     for i in range(output_dim):
58         for j in range(output_dim):
59             region = X[i*stride:i*stride+pool_size, j*stride:j*stride+
60                         pool_size]
61             if pool_type == "max":
62                 output[i, j] = np.max(region)
63             elif pool_type == "average":
64                 output[i, j] = np.mean(region)
65             else:
66                 raise ValueError("Pooling type must be 'max' or 'average'")
67
68     return output
69
70
71 # Example Input Matrix X and Filter W
72 X = np.array([

```

```

63 [4, 9, 2, 5, 8, 3],
64 [2, 4, 0, 4, 0, 3],
65 [2, 5, 4, 5, 4, 5],
66 [5, 5, 4, 7, 8, 2],
67 [5, 7, 9, 2, 1, 9],
68 [5, 5, 8, 3, 8, 4]
69 ])
70
71 W = np.array([
72     [1, 0, -1],
73     [1, 0, -1],
74     [1, 0, -1]
75 ])
76
77 # Perform Convolution
78 conv_result = convolution_2d(X, W, stride=2, padding="same")
79 print("Convolution Result with Padding:")
80 print(pd.DataFrame(conv_result))
81
82 # Perform Pooling
83 pooled_result = pooling_2d(conv_result, pool_size=2, pool_type="max",
84     stride=2)
85 print("\nMax Pooling Result:")
86 print(pd.DataFrame(pooled_result))

```

## 7.6.9 Fully Connected Layers

After several convolutional and pooling layers, Convolutional Neural Networks typically include fully connected layers that integrate the extracted spatial features to produce the final output. Fully connected layers are particularly useful in classification tasks where each node in the output layer corresponds to a class.

## 7.6.10 Training Convolutional Neural Networks

Convolutional Neural Networks are trained using supervised learning techniques, often involving the following steps:

- **Forward Propagation:** The input data is passed through the network to compute the output.
- **Loss Calculation:** A loss function calculates the error between the network's prediction and the actual target.
- **Backpropagation:** The gradients of the loss with respect to each weight and filter are computed to update parameters.
- **Optimization:** Optimization algorithms, such as stochastic gradient descent or Adam, are applied to minimize the loss and improve the network's performance.

### 7.6.11 Applications of Convolutional Neural Networks

Convolutional Neural Networks are highly versatile and widely used across various domains, including:

- **Image Classification:** Identifying objects in images and categorizing them into pre-defined classes.
- **Object Detection:** Locating and classifying objects within an image, often used in autonomous vehicles and surveillance systems.
- **Image Segmentation:** Dividing an image into meaningful segments or regions, widely used in medical imaging.
- **Natural Language Processing:** Applications in text classification, sentence analysis, and even text-based generative models.

### 7.6.12 Challenges and Limitations

Despite their powerful feature extraction capabilities, Convolutional Neural Networks also have limitations:

- **Computational Complexity:** Convolutional Neural Networks are resource-intensive, requiring substantial computational power and memory, especially for large networks.
- **Dependence on Large Datasets:** Convolutional Neural Networks generally require large labeled datasets to generalize effectively.
- **Vulnerability to Adversarial Attacks:** Convolutional Neural Networks can be sensitive to adversarial attacks, where small perturbations to the input data can significantly affect predictions.

### 7.6.13 Summary

Convolutional Neural Networks represent a **groundbreaking approach in machine learning**, providing robust and scalable methods for extracting features from structured data like images. By leveraging convolutional and pooling layers, Convolutional Neural Networks enable effective handling of complex image processing tasks. They form the backbone of many applications in computer vision, setting a foundation for advancements in autonomous systems, medical imaging, and beyond.

## 7.7 Recurrent Neural Networks

Recurrent Neural Networks are a class of artificial neural networks designed to **model sequential data by maintaining a memory of previous inputs**. Unlike traditional feed-forward neural networks, Recurrent Neural Networks have **loops** in their architecture, allowing information to persist and propagate through time. This makes RNNs particularly effective for tasks where the order and context of input data are crucial, such as time series forecasting, natural language processing, and speech recognition.

### 7.7.1 Architecture of Recurrent Neural Networks

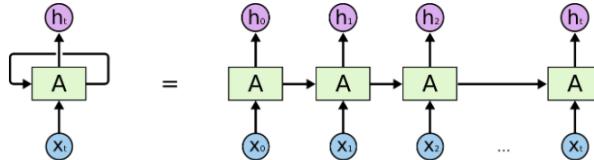


Figure 7.3: An unrolled recurrent neural network

The key distinguishing feature of a Recurrent Neural Network is the presence of **recurrent connections**, where the output of a neuron at time step  $t$  is fed back into the network as input at the next time step  $t+1$ . This feedback loop allows the network to maintain a memory of previous inputs, making it suitable for sequential data processing. The architecture of a basic RNN can be represented as follows:

$$h_t = \sigma(W_{hh}h_{t-1} + W_{hx}x_t + b)$$

where:

- $h_t$  is the **hidden state** at time step  $t$ ,
- $W_{hh}$  is the **weight matrix** for the hidden state at the previous time step,
- $W_{hx}$  is the **weight matrix** for the input at time step  $t$ ,
- $x_t$  is the input at time step  $t$ ,
- $b$  is the **bias term**, and
- $\sigma$  is the **activation function** (typically the tanh function or ReLU).

The output at each time step is typically computed as:

$$y_t = W_{hy}h_t + c$$

where:

- $y_t$  is the output at time step  $t$ ,
- $W_{hy}$  is the weight matrix between the hidden state and the output,
- $c$  is the output bias.

### 7.7.2 Training Recurrent Neural Networks

Recurrent Neural Networks are trained using supervised learning, typically using **backpropagation through time (BPTT)** or the more efficient truncated BPTT for long sequences. The backpropagation algorithm computes the gradients of the loss with respect to each weight, which are used to update the parameters of the network. The challenge of training RNNs arises from the problem of vanishing and exploding gradients, which can hinder the learning process, particularly in long sequences.

### 7.7.3 The Problem of Long-Term Dependencies

One of the primary appeals of Recurrent Neural Networks (RNNs) lies in their potential to **connect past information to current tasks**. This feature makes them particularly well-suited for **sequential data processing, such as video analysis or language modeling**. For instance, prior frames in a video sequence might help inform the understanding of the present frame. If RNNs could consistently establish such connections, they would be incredibly powerful tools for handling complex temporal relationships. But how effectively can they do this in practice? It depends on the context.

#### 7.7.3.1 Short-Term Dependencies

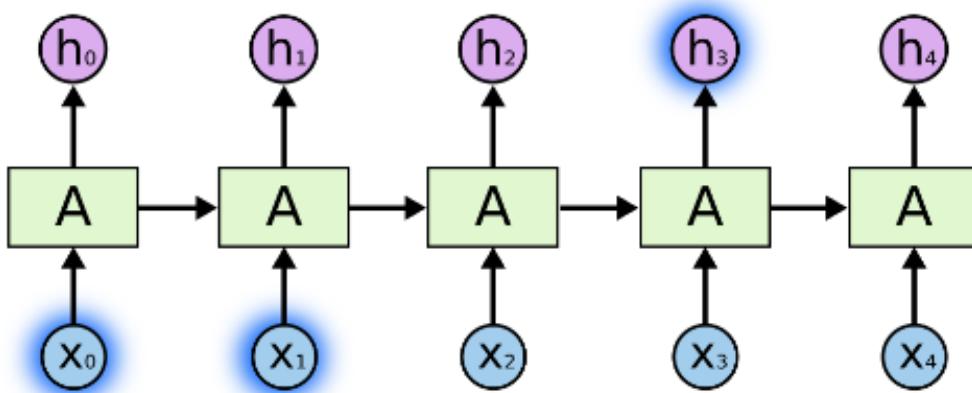


Figure 7.4: Short-Term Dependencies

In many tasks, only recent information is required to make accurate predictions. For example, in a **language modeling task**, predicting the next word in the phrase “*The clouds are in*

*the sky*" relies on the immediate context. Here, the next word (*sky*) is obvious, and the gap between relevant information and its application is minimal. In such cases, RNNs can efficiently learn to use past data to make predictions.

### 7.7.3.2 Long-Term Dependencies

However, not all tasks are this straightforward. In some situations, critical information lies much further back in the sequence. For example, consider the sentence "*I grew up in France... I speak fluent French.*" While the recent context suggests the next word might be the name of a language, determining the correct language (*French*) requires retrieving information from the distant past (*France*).

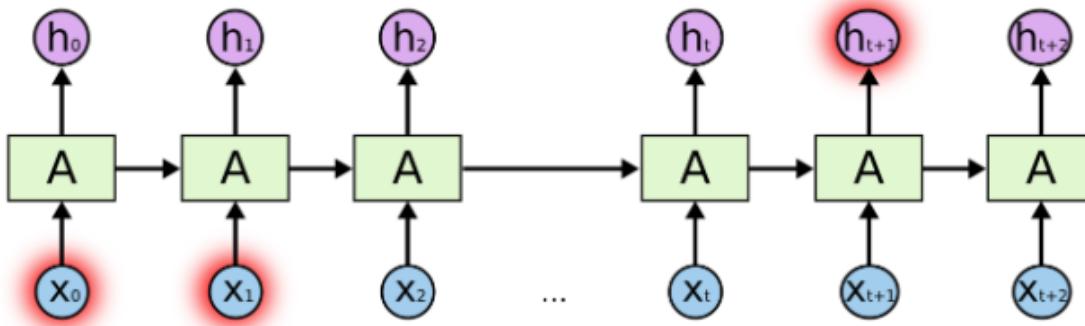


Figure 7.5: Long-Term Dependencies

As the gap between the relevant information and its application grows, RNNs struggle to establish these connections. This phenomenon, known as the problem of long-term dependencies, arises from the inherent challenges of training RNNs on extended sequences.

### 7.7.3.3 Why Do RNNs Struggle?

In **theory**, RNNs are capable of **learning long-term dependencies**. It is possible to manually configure their parameters to solve toy problems that require such connections. However, in **practice**, RNNs often **fail to learn these relationships during training**. Research by Hochreiter (1991) and Bengio et al. (1994) identified fundamental issues such as the **vanishing gradient problem**, which causes gradients to diminish exponentially as they propagate backward through time. This makes it difficult for the model to update parameters related to distant information, effectively "forgetting" critical long-term dependencies.

### 7.7.3.4 Overcoming the Problem

Fortunately, advances in neural network architecture, such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), have addressed this limitation. By

introducing mechanisms like forget gates and memory cells, these models are specifically designed to retain relevant information over longer time spans, effectively solving the long-term dependency problem. As a result, LSTMs and GRUs have become the preferred choice for tasks involving sequences with extended temporal dependencies.

#### 7.7.4 Long Short-Term Memory (LSTM) Networks

One of the **key advancements** in RNNs is the development of Long Short-Term Memory (LSTM) networks, which are designed to mitigate the vanishing gradient problem. LSTMs introduce memory cells that can maintain information over long periods of time, enabling the network to remember long-range dependencies. An LSTM unit consists of several components:

- **Forget Gate:** Decides which information from the previous time step should be discarded from the memory.
- **Input Gate:** Determines which new information should be stored in the memory.
- **Cell State:** The long-term memory that retains important information over time.
- **Output Gate:** Determines which information should be outputted from the memory.

These components allow LSTMs to overcome the vanishing gradient problem and capture long-range dependencies more effectively than vanilla RNNs.

#### 7.7.5 Applications of Recurrent Neural Networks

Recurrent Neural Networks are widely used in applications where data has temporal or sequential dependencies. Some common applications include:

- **Natural Language Processing:** RNNs are used in tasks such as machine translation, sentiment analysis, and speech recognition, where the order of words or characters matters.
- **Time Series Forecasting:** RNNs are effective for predicting future values in time series data, such as stock market prices, weather predictions, and sales forecasting.
- **Speech Recognition:** RNNs can process sequential acoustic signals and convert them into text.
- **Music Generation:** RNNs are used to generate sequences of musical notes, allowing for the creation of melodies and harmonies.

### 7.7.6 Challenges and Limitations

Despite their advantages, Recurrent Neural Networks have several limitations:

- **Vanishing and Exploding Gradients:** During training, the gradients of the loss function can either shrink to zero (vanish) or grow exponentially (explode), making it difficult to train RNNs on long sequences.
- **Long-Term Dependencies:** While LSTMs and Gated Recurrent Units (GRUs) address the issue of vanishing gradients, RNNs still face challenges in learning long-term dependencies across many time steps.
- **Computational Complexity:** Training RNNs can be computationally expensive, particularly for large datasets or long sequences.

### 7.7.7 Gated Recurrent Units (GRUs)

Gated Recurrent Units are another **variant** of RNNs, designed to **simplify the architecture** of LSTMs while maintaining their ability to model **long-range dependencies**. GRUs combine the forget and input gates into a single update gate, reducing the complexity of the model while still capturing important sequential patterns. GRUs have been shown to perform comparably to LSTMs on many tasks, with fewer parameters and faster training times.

### 7.7.8 Summary

Recurrent Neural Networks represent a powerful approach to modeling **sequential data**, with applications ranging from **natural language processing** to **time series forecasting**. While traditional RNNs struggle with issues like **vanishing gradients**, advancements such as Long Short-Term Memory networks and Gated Recurrent Units have improved the ability of RNNs to capture long-term dependencies. Despite **challenges**, RNNs continue to be a fundamental tool in machine learning, with ongoing research focused on improving their scalability and efficiency in handling complex sequential tasks.

### 7.7.9 Distinguishing between Recurrent Neural Networks and Recursive Neural Networks

Recurrent Neural Networks and Recursive Neural Networks are both specialized types of neural networks, designed to handle different structures of data. While they share some similarities in their ability to process complex information, they differ significantly in their approach and applications.

#### Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are designed to process sequential data by maintaining a hidden state, which is updated at each time step based on the input and the previous hidden state. This hidden state acts as **a form of memory**, allowing RNNs to capture **temporal dependencies** in data. RNNs are particularly useful for tasks involving time series, language modeling, and speech recognition, where the order of inputs and previous context is crucial.

Mathematically, the process in RNNs can be described as:

$$h_t = \sigma(W_{hh}h_{t-1} + W_{hx}x_t + b)$$

where  $h_t$  is the hidden state at time step  $t$ , and the network updates its memory as new data  $x_t$  is processed.

### Recursive Neural Networks

In contrast, Recursive Neural Networks are used to process **hierarchical or tree-like data structures**, such as **parse trees in natural language processing** or **tree-structured data in computer vision**. Recursive networks do not process **sequential data** in a temporal fashion, but instead apply operations recursively over the nodes of a tree. The structure of the network allows it to learn dependencies within complex hierarchical data, making it ideal for tasks like syntactic parsing or modeling relationships in structured graphs.

Recursive networks apply a function  $f$  recursively to nodes  $v$  and their child nodes:

$$h_v = f(h_{v_1}, h_{v_2}, \dots, h_{v_k})$$

where  $v_1, v_2, \dots, v_k$  are the child nodes of node  $v$ , and  $f$  is a function that processes the node's children and generates its representation.

### Key Differences

The main differences between **RNNs** and **Recursive Neural Networks** are in the type of data they process and their architecture:

- **Data Structure:** RNNs are designed to **handle sequential data**, where each input is processed in a **linear fashion**, while Recursive Neural Networks are optimized for **hierarchical data**, where relationships between elements are structured in trees.
- **Processing Method:** RNNs **update their hidden state** at each time step based on previous inputs and states, making them suitable for tasks with **temporal dependencies**. Recursive Neural Networks, on the other hand, apply recursive operations over tree-like structures, processing data based on parent-child relationships.
- **Memory and Context:** RNNs maintain a **hidden state** that evolves with time, storing **contextual information** as new data is processed. Recursive Neural Networks build representations of the data hierarchically, storing context in the form of nodes and their relationships.

### Applications

Recurrent Neural Networks are particularly effective for tasks where the sequence and order of data matter, such as:

- Time series forecasting
- Speech recognition
- Language modeling and machine translation
- Sentiment analysis and text classification

Meanwhile, recursive Neural Networks are most suitable for tasks involving structured or hierarchical data, such as:

- Syntactic parsing in natural language processing
- Semantic parsing and understanding
- Tree-structured data analysis in computer vision
- Graph-based models and relation extraction

## Conclusion

In summary, while both Recurrent and Recursive Neural Networks share the ability to process complex data, they are tailored to different types of input structures. Recurrent Neural Networks excel at handling sequential data with temporal dependencies, whereas Recursive Neural Networks are specialized in managing hierarchical or tree-based data. Understanding these differences allows for the appropriate choice of network depending on the specific problem and data structure.

## 7.8 Long Short-Term Memory Networks

Long Short-Term Memory (LSTM) networks are a special kind of Recurrent Neural Network (RNN) designed to **address the problem of vanishing gradients**, which makes traditional RNNs ineffective in learning **long-range dependencies**. LSTMs were introduced by Hochreiter and Schmidhuber in 1997 and have since become one of the most widely used types of neural networks for sequential data.

### 7.8.1 Architecture of LSTM

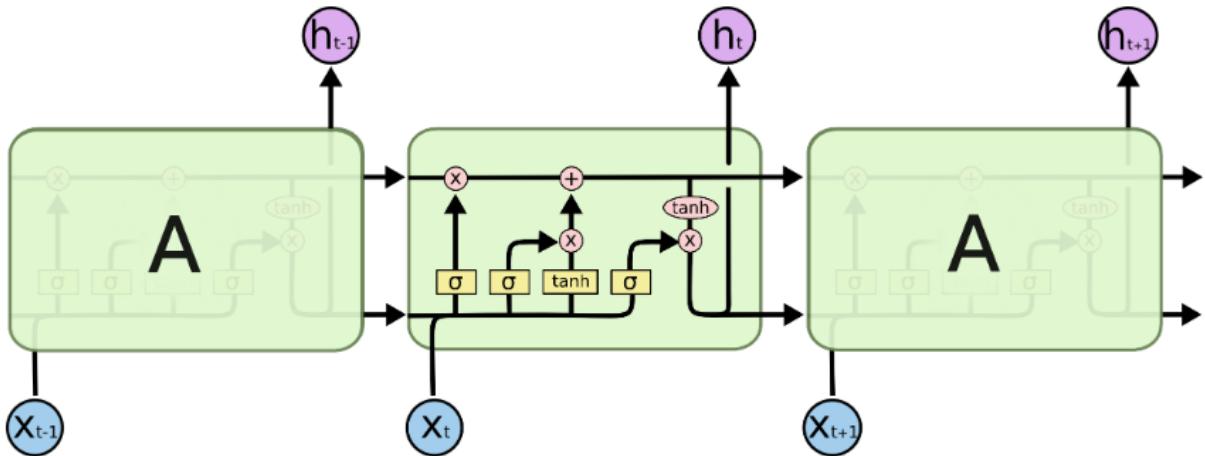


Figure 7.6: LSTM architecture

The **key innovation** of LSTM networks lies in their **memory cell**, which is capable of maintaining information for **long periods of time**. An **LSTM unit** consists of **three gates** that control the flow of information: the input gate, the forget gate, and the output gate. These gates allow the LSTM to selectively store, update, and output information, making it more effective at capturing long-term dependencies.

#### 7.8.1.1 Memory Cell

The memory cell is at the **core of the LSTM**. It acts as a “conveyor belt” for information, carrying information throughout the network. The **cell state** is modified by the input, forget, and output gates at each time step.

#### 7.8.1.2 Gates in LSTM

1. **Forget Gate ( $f_t$ )**: Determines which information from the previous time step should be discarded. The **forget gate** is a **sigmoid activation function** applied to the **concatenation** of the **previous hidden state** and the **current input**. Its output is a value between 0 and 1, representing the fraction of the previous memory to retain.

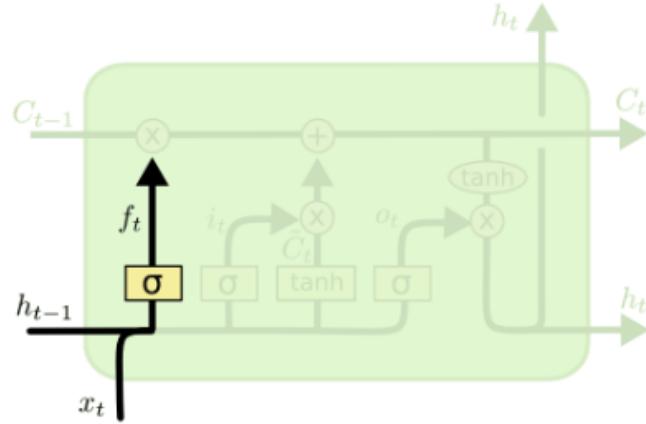


Figure 7.7: Forget Gate

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where  $\sigma$  is the sigmoid function,  $h_{t-1}$  is the previous hidden state,  $x_t$  is the current input, and  $W_f$  and  $b_f$  are the weights and biases associated with the forget gate.

**2. Input Gate ( $i_t$ ):** Controls **how much of the new information should be added** to the memory cell. The **input gate** is also a **sigmoid function** applied to the previous hidden state and the current input. Additionally, a **candidate memory state**,  $\tilde{C}_t$ , is generated using a hyperbolic tangent activation function.

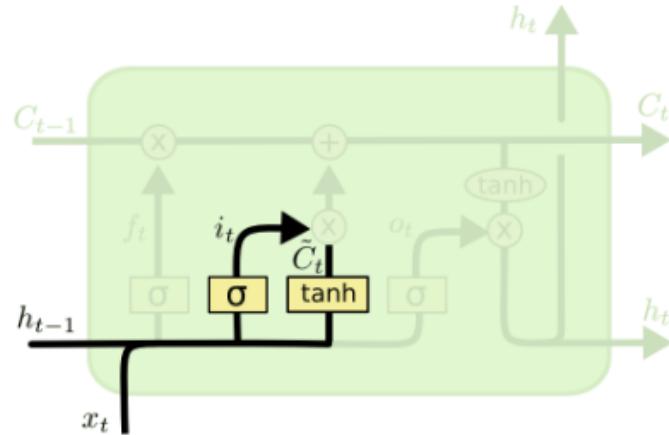


Figure 7.8: Input Gate

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

The candidate memory state is used to update the memory cell, depending on the output of the input gate.

3. **Output Gate** ( $o_t$ ): Determines what part of the memory cell should be output to the hidden state. The **output gate** is a sigmoid function applied to the previous hidden state and the current input, modulated by the current memory state.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

#### 7.8.1.3 Updating the Memory Cell

The **memory cell state** is updated using the forget gate, the input gate, and the candidate memory state:

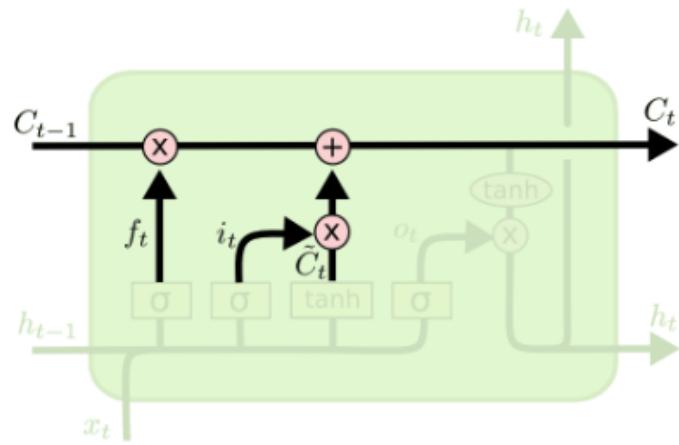


Figure 7.9: Update memory cell state

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

where  $C_t$  is the **updated memory cell state**,  $C_{t-1}$  is the previous memory state, and  $\tilde{C}_t$  is the candidate memory state.

Finally, the hidden state  $h_t$  is updated as:

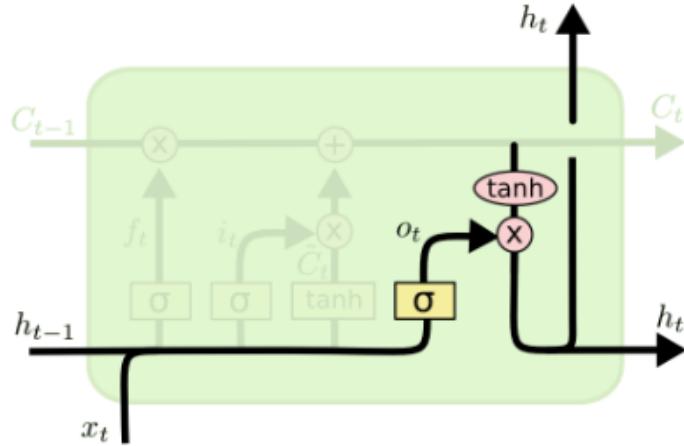


Figure 7.10: Update hidden state

$$h_t = o_t \cdot \tanh(C_t)$$

This allows the network to selectively output relevant information at each time step.

### 7.8.2 Advantages of LSTM

LSTMs **address the vanishing gradient problem** by using their gating mechanisms to regulate the flow of information. This allows LSTMs to **learn long-term dependencies** more effectively than traditional RNNs. Other advantages include:

- **Long-term memory retention:** The memory cell can store information over long periods, enabling the model to remember information from previous time steps and use it to make better predictions.
- **Selective forgetting:** The forget gate allows LSTMs to selectively forget irrelevant information, improving the model's ability to focus on important data.
- **Flexible learning:** LSTMs can adaptively learn the appropriate degree of memory retention and information flow, making them more versatile across different tasks.

### 7.8.3 Applications of LSTM Networks

LSTMs have been successfully applied in various fields, particularly where long-range dependencies are important. Some of the key applications include:

- **Speech recognition:** LSTMs are used to model sequential speech data, which requires remembering context over time to transcribe speech accurately.

- **Natural language processing:** LSTMs are widely used for tasks such as machine translation, text generation, and sentiment analysis, where understanding context over time is crucial.
- **Time series forecasting:** LSTMs excel in modeling sequential time series data, such as stock prices, weather patterns, and sensor data, where long-term dependencies are often present.
- **Video analysis:** LSTMs are used in video classification and action recognition, where the temporal sequence of frames is critical to understanding the content.

#### 7.8.4 Conclusion

Long Short-Term Memory networks represent a powerful advancement over traditional RNNs, enabling models to effectively capture long-term dependencies and handle complex sequential data. With their gating mechanisms and memory cells, LSTMs are well-suited for a wide range of applications, from natural language processing to time series forecasting and speech recognition. Despite their complexity, LSTMs remain a cornerstone of modern deep learning techniques for sequence modeling.

## 7.9 Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is a type of Recurrent Neural Network (RNN) architecture that was introduced by Cho et al. in 2014. It is designed to **solve the vanishing gradient problem** of traditional RNNs, **similar to Long Short-Term Memory (LSTM) networks, but with a simpler architecture**. GRUs have become widely adopted due to their **computational efficiency** and effectiveness in **handling sequential data**.

### 7.9.1 Architecture of GRU

GRUs combine the **forget and input gates** of LSTM networks into a single update gate and use a reset gate to control how much of the previous hidden state should be forgotten. This results in fewer parameters and a simpler structure, making them faster to train while still maintaining the ability to capture long-range dependencies.

#### 7.9.1.1 Gates in GRU

1. **Update Gate ( $z_t$ )**: The update gate determines the degree to which the previous hidden state should be carried forward into the current time step. It is a **sigmoid activation** function applied to the concatenation of the previous hidden state and the current input.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

where  $\sigma$  is the sigmoid function,  $h_{t-1}$  is the previous hidden state,  $x_t$  is the current input, and  $W_z$  and  $b_z$  are the weights and biases associated with the update gate.

2. **Reset Gate ( $r_t$ )**: The reset gate controls **how much of the previous hidden state should be ignored**. It is also a sigmoid function applied to the previous hidden state and the current input.

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

3. **Candidate Hidden State ( $\tilde{h}_t$ )**: The candidate hidden state represents the **potential new hidden state**, which is generated based on the current input and the reset gate. This state is computed using a tanh activation function.

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \cdot h_{t-1}, x_t] + b_h)$$

#### 7.9.1.2 Updating the Hidden State

The hidden state is updated by a linear combination of the previous hidden state and the candidate hidden state, weighted by the update gate:

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t$$

The update gate allows the GRU to decide how much of the previous hidden state should be retained and how much of the candidate hidden state should be adopted.

## 7.9.2 Advantages of GRU

GRUs offer several advantages over traditional RNNs and LSTMs:

- **Simpler architecture:** By combining the forget and input gates into one update gate, GRUs have fewer parameters than LSTMs, making them computationally more efficient while maintaining a similar level of performance.
- **Faster training:** With fewer parameters, GRUs typically train faster than LSTMs, making them a suitable choice for large-scale applications or time-sensitive tasks.
- **Effective at learning long-range dependencies:** GRUs, like LSTMs, address the vanishing gradient problem, making them capable of learning long-term dependencies in sequential data.

## 7.9.3 Applications of GRU

GRUs are widely used in tasks that involve sequential data, similar to LSTMs. Some of the most common applications include:

- **Natural language processing:** GRUs are frequently used in tasks such as machine translation, language modeling, and sentiment analysis, where context over time is important.
- **Speech recognition:** GRUs can model sequential speech data, where it is necessary to remember context from previous speech frames to make accurate predictions.
- **Time series forecasting:** GRUs are well-suited for predicting future values based on historical data, such as stock prices, sensor data, and weather forecasting.
- **Video and action recognition:** GRUs can be applied in analyzing video sequences to recognize actions or classify video content based on temporal dependencies.

## 7.9.4 Comparison to LSTM

While both GRUs and LSTMs address the vanishing gradient problem and are effective at capturing long-range dependencies, there are key differences:

- **Gate structure:** GRUs combine the forget and input gates into a single update gate, while LSTMs use separate gates for forget, input, and output.
- **Complexity:** GRUs have fewer parameters and a simpler structure than LSTMs, which makes them computationally more efficient.
- **Performance:** In many cases, GRUs perform similarly to LSTMs, but LSTMs may be better at handling more complex tasks due to their greater flexibility.

### 7.9.5 Conclusion

Gated Recurrent Units (GRUs) are an efficient and effective alternative to Long Short-Term Memory (LSTM) networks. By simplifying the gating mechanisms, GRUs offer a faster and less computationally expensive way to capture long-range dependencies in sequential data. Although GRUs may not have the same level of flexibility as LSTMs, they perform well across many applications, including natural language processing, speech recognition, and time series forecasting. As such, they remain an important architecture in the field of deep learning.

## 7.10 Bidirectional Long Short-Term Memory Networks

Bidirectional Long Short-Term Memory (**BiLSTM**) networks are an **extension** of the Long Short-Term Memory (**LSTM**) architecture, designed to **capture dependencies in sequential data** by processing information in **both forward and backward directions**. This bidirectional processing capability enables the network to have a complete view of the input sequence at any given time step, making it particularly valuable for tasks where understanding the context around each element is essential.

### 7.10.1 Architecture of BiLSTM

The architecture of a BiLSTM network consists of **two LSTM layers** that process the input sequence in **opposite directions**. The first LSTM layer processes the sequence in the forward direction, from the first element to the last, capturing the **contextual information leading up** to each time step. The second LSTM layer processes the sequence in reverse, from the last element to the first, capturing the **information following** each time step. The **outputs** from both LSTM layers are then **concatenated or otherwise combined to form a final representation** that integrates **both past and future context** around each element in the sequence.

### 7.10.2 Advantages of BiLSTM

The **primary advantage** of BiLSTM networks over unidirectional LSTMs is their ability to access information from **both past and future time** steps, which is critical for certain tasks in natural language processing and other sequential data contexts. For example:

- **Contextual Understanding:** In language-related tasks, such as named entity recognition or sentiment analysis, the meaning of a word or phrase can depend on both previous and subsequent words. BiLSTMs allow the network to incorporate full contextual information, improving performance on these tasks.
- **Improved Performance on Sequential Tasks:** For tasks such as speech recognition, where the meaning of sounds can depend on sounds that come before and after, BiLSTMs can provide more accurate predictions by capturing dependencies in both temporal directions.
- **Enhanced Representation Power:** By considering both directions, BiLSTMs can form richer representations of each input element, which is beneficial for complex tasks such as machine translation, text generation, and time-series prediction.

### 7.10.3 Applications of BiLSTM Networks

BiLSTM networks are widely used across various domains due to their ability to model complex, **bidirectional dependencies**. Some notable applications include:

- **Natural Language Processing (NLP)**: BiLSTMs are commonly used in NLP tasks like part-of-speech tagging, named entity recognition, and question answering, where understanding the context of words from both directions enhances model performance.
- **Speech and Audio Processing**: BiLSTMs are employed in speech recognition and other audio analysis tasks where temporal dependencies in both directions contribute to better understanding and classification of sounds.
- **Biomedical Signal Processing**: In medical research, BiLSTMs have been used for analyzing sequential data such as electrocardiograms (ECGs) and other time-series physiological data, where understanding patterns from both past and future observations can provide crucial insights.

### 7.10.4 Comparison with Unidirectional LSTM

Compared to standard unidirectional LSTMs, BiLSTMs offer **increased accuracy on tasks requiring bidirectional context**. However, they are computationally more expensive since they require running two LSTM layers instead of one. Despite this added cost, the improvement in representational capability often justifies their use in performance-sensitive applications.

In summary, BiLSTM networks extend the capabilities of traditional LSTMs by incorporating information from both past and future time steps, which is essential for tasks with bidirectional dependencies. This dual-contextual processing makes BiLSTM networks a powerful tool in many sequence modeling applications.

### 7.10.5 LSTM Equations

Bidirectional Long Short-Term Memory (**BiLSTM**) networks build on the foundational equations of Long Short-Term Memory (**LSTM**) networks by incorporating a second, reverse LSTM layer. This architecture allows information to flow in both forward and backward directions, which enables the network to capture dependencies in the data that occur both before and after each time step. In next sections, we delve into the mathematics underlying BiLSTM networks. An LSTM cell comprises several gates including **input**, **forget**, and **output gates** that control the flow of information. These gates are calculated as follows:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad (7.1)$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad (7.2)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad (7.3)$$

where:

- $f_t$  is the **forget gate**, which decides what information to discard from the cell state.
- $i_t$  is the **input gate**, which determines what new information to add to the cell state.
- $o_t$  is the **output gate**, which controls the amount of cell state information sent to the hidden state.
- $W$  and  $U$  **matrices** and  $b$  vectors are **learned parameters**.
- $\sigma$  is the **sigmoid activation** function.

The cell state  $C_t$  and hidden state  $h_t$  are updated as follows:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tanh(W_c x_t + U_c h_{t-1} + b_c) \quad (7.4)$$

$$h_t = o_t \odot \tanh(C_t) \quad (7.5)$$

These equations govern the information flow within a **single LSTM cell** in the forward direction. For BiLSTM networks, a second LSTM cell processes the sequence in the reverse direction.

### 7.10.6 Bidirectional Processing

In BiLSTM networks, each input  $x_t$  is processed by **two LSTM layers**: one moving **forward** through time (from  $x_1$  to  $x_T$ ) and another moving **backward** (from  $x_T$  to  $x_1$ ). This results in **two hidden states** at each time step, one from the forward LSTM  $h_t^\rightarrow$  and one from the backward LSTM  $h_t^\leftarrow$ .

The **final output**  $h_t$  at each time step  $t$  is typically a **concatenation** of the forward and backward hidden states:

$$h_t = [h_t^\rightarrow; h_t^\leftarrow] \quad (7.6)$$

Thus, the **combined** hidden state  $h_t$  captures information from **both past and future contexts**, enhancing the network's ability to **model dependencies** in both directions.

### 7.10.7 Computational Complexity of BiLSTM

BiLSTM networks **double** the number of **computations** compared to unidirectional LSTM networks, as each time step requires processing in two directions. However, the ability to capture **bidirectional dependencies** often justifies the increased computational load, particularly for tasks where context from both past and future time steps is essential.

### 7.10.8 Gradient Flow and Training

During training, gradients in BiLSTM networks are computed for both the forward and backward LSTM layers. This **dual-gradient computation** allows the network to optimize parameters in both directions simultaneously. The concatenation of forward and backward hidden states also helps in reducing the vanishing gradient problem, as each LSTM layer only needs to propagate gradients in one direction, maintaining more stable gradient magnitudes across time steps.

### 7.10.9 Applications of BiLSTM Networks in Sequence Modeling

The mathematical foundation of BiLSTM networks makes them highly effective in **sequence modeling tasks** where context from both directions matters. The ability to **incorporate information from both the past and the future** enables BiLSTM networks to perform well on tasks like named entity recognition, language translation, and speech recognition, where capturing **bidirectional dependencies** is crucial.

In summary, BiLSTM networks build on the LSTM framework by adding a backward processing layer, allowing the model to retain information from both past and future contexts. This dual-context approach provides enhanced representation capabilities, making BiLSTM a powerful model for sequence modeling applications.

### 7.10.10 BiLSTM Algorithm

Given an input sequence  $X = [x_1, x_2, \dots, x_T]$  where  $T$  is the sequence length, the algorithm for a BiLSTM is as follows:

#### 1. Initialize Parameters:

- Initialize two separate LSTM layers: one for processing the sequence in the forward direction (left-to-right) and one for the backward direction (right-to-left).
- Let  $\overrightarrow{\text{LSTM}}$  represent the forward LSTM and  $\overleftarrow{\text{LSTM}}$  represent the backward LSTM.

#### 2. Forward Pass in Forward LSTM:

- For each time step  $t$  from 1 to  $T$ :

$$\overrightarrow{h}_t, \overrightarrow{c}_t = \overrightarrow{\text{LSTM}}(x_t, \overrightarrow{h}_{t-1}, \overrightarrow{c}_{t-1}) \quad (7.7)$$

- The forward LSTM updates the **hidden state**  $\overrightarrow{h}_t$  and **cell state**  $\overrightarrow{c}_t$  using standard LSTM update equations for the input, forget, output, and cell gates.

#### 3. Backward Pass in Backward LSTM:

- For each time step  $t$  from  $T$  to 1:

$$\overleftarrow{h}_t, \overleftarrow{c}_t = \overleftarrow{\text{LSTM}}(x_t, \overleftarrow{h}_{t+1}, \overleftarrow{c}_{t+1}) \quad (7.8)$$

- The backward LSTM follows standard LSTM equations but in reverse, processing the sequence from the last element to the first.

#### 4. Concatenate Forward and Backward States:

- For each time step  $t$ , concatenate the forward and backward hidden states to form the final BiLSTM hidden state:

$$h_t = [\overrightarrow{h}_t; \overleftarrow{h}_t] \quad (7.9)$$

- This concatenated hidden state  $h_t$  captures context from both past and future.

#### 5. Output Layer

- Depending on the application, the combined hidden state  $h_t$  can be passed through an output layer, such as a fully connected layer with softmax activation for classification.

## 7.11 Fundamental of Embeddings

### 7.11.1 What is Embedding?

Embedding is a means of representing objects like text, images, and audio as points in a continuous vector space, where the locations of these points are semantically meaningful to machine learning (ML) algorithms.

Embedding is a critical tool for ML engineers who build applications like text search engines, recommendation systems, and AI-generated text detection systems. In essence, embeddings enable machine learning models to find similar objects efficiently.

Unlike traditional ML techniques, embeddings are learned from data using algorithms like neural networks. This allows models to identify complex patterns in data, which would otherwise be difficult to discern manually.

In the context of detecting AI-generated text, embeddings can be used to differentiate between human-written and machine-generated content. By converting text into embeddings, a model can detect subtle differences in writing patterns, structure, and word usage that might indicate whether a text was produced by an AI.

### 7.11.2 How Embedding Works

Most machine learning algorithms require numerical data as input. Therefore, data needs to be converted into a numerical format, such as:

- Creating a bag-of-words representation for text.
- Converting entire paragraphs into dense embeddings using models like BERT.

Objects processed by an embedding model are represented as **vectors**. A vector is an array of numbers (e.g.,  $[0.76, -0.25, \dots, 0.91]$ ), where each number indicates a feature. The **similarity between embeddings is often measured using metrics like cosine similarity or Euclidean distance**.

To detect AI-generated text, embeddings of text samples can be compared. If the embeddings of a given document are closer to those typically generated by a known LLM (e.g., GPT-3), it might suggest that the content is AI-generated.

#### 7.11.2.1 Word2Vec Example

Word2Vec, developed by Google in 2013, efficiently creates word embeddings using a two-layer neural network. It converts words into n-dimensional vectors. For example:

- "essay" =  $[0.562, -0.235, \dots, 0.879]$

- "thesis" = [0.542, 0.125, ..., 0.654]
- "article" = [0.632, -0.154, ..., 0.798]

In the context of detecting AI-generated text, we can use Word2Vec embeddings to identify subtle linguistic patterns that differentiate machine-generated text from human-written content.

### **Example: Identifying Overused Transitions and Patterns**

AI-generated text often tends to repeat specific transition words or phrases more frequently than human writers. For instance, AI models may overuse phrases like "`in conclusion`" or "`therefore`" to connect ideas. Using Word2Vec, we can identify this pattern by examining the embeddings of transition words and comparing their usage frequencies:

- First, train a Word2Vec model on a mixed dataset of AI-generated and human-written essays.
- Generate word embeddings for frequently used transitions (e.g., "`thus`", "`moreover`", "`furthermore`").
- Calculate the cosine similarity between these words to detect clusters.

**Practical Example** Suppose we have two text samples:

1. Human-written: *"The results indicate a strong correlation. Thus, the hypothesis is supported."*
2. AI-generated: *"The data shows a significant trend. Thus, it can be concluded that the pattern is consistent. Thus, we observe this effect repeatedly."*

By analyzing the embeddings of the word "`thus`" across these samples, we may notice that AI-generated text tends to place this word closer in context to adjacent sentences, indicating repetitive usage.

Using the Word2Vec embeddings, we can build a classifier that detects overuse of transition words by calculating:

$$\text{Similarity Score} = \text{cosine\_similarity}(\text{transition\_word\_embedding}, \text{context\_embedding})$$

If the similarity score is high across many occurrences of the same transition word, the text is more likely to be AI-generated.

#### **7.11.2.2 Recommendation Embedding**

Recommendation systems use embeddings to represent users and items as high-dimensional vectors. The dot product of user and item embeddings correlates with the user's preference:

$$\text{Score} = \text{User Embedding} \cdot \text{Item Embedding}$$

In detecting AI-generated text, embeddings can be used to compare a document's style with a database of known AI-generated texts. If the embedding of a new text has a high similarity score with known AI-generated texts, it may indicate that the text is machine-generated.

### 7.11.3 Types of Embeddings

**Word Embeddings** Word embeddings capture the semantic relationships and contextual meanings of words based on their usage patterns in large text corpora. Unlike traditional representations like one-hot encoding, which are sparse and lack context, word embeddings use dense vectors to capture the meaning of words in a way that reflects their semantic relationships. Popular word embedding models include:

- **Word2Vec**: This model represents words as vectors based on the context in which they appear. It uses techniques like Continuous Bag of Words (CBOW) and Skip-gram to predict words based on their neighbors. Word2Vec can capture analogies such as "king" - "man" + "woman" = "queen", illustrating its ability to understand word relationships.
- **GloVe (Global Vectors for Word Representation)**: GloVe uses word co-occurrence statistics across a corpus to create embeddings that capture both local context and global statistics. It is effective in capturing semantic similarity, such as grouping words like "dog" and "cat" closer than "dog" and "car".
- **FastText**: Unlike Word2Vec, FastText breaks words into subword components (character n-grams) to create embeddings. This allows it to handle out-of-vocabulary words and improve representation for morphologically rich languages.
- **Transformer-based models (BERT, GPT)**: These models generate contextual word embeddings, meaning that the same word can have different embeddings based on the context it appears in. For instance, the word "bank" in "river bank" versus "financial bank" will have different embeddings in BERT.

In detecting AI-generated text, word embeddings can help identify patterns in word usage. For example, AI models may overuse certain connectors (like "hence" or "therefore") or use rare synonyms that humans typically avoid. By comparing the embeddings of specific words used in an essay, it is possible to flag text that appears suspiciously non-human.

**Text Embeddings** Text embeddings extend the concept of word embeddings to represent entire sentences, paragraphs, or documents. These embeddings capture the meaning of the text as a whole, rather than focusing on individual words. Common models for generating text embeddings include:

- **Doc2Vec**: An extension of Word2Vec that creates embeddings for entire documents. It captures contextual information beyond individual words, allowing for semantic similarity searches and document classification.

- **Universal Sentence Encoder (USE)**: This model focuses on generating embeddings that capture the meaning of entire sentences, making it ideal for tasks like sentence similarity, clustering, and text classification.
- **BERT (Bidirectional Encoder Representations from Transformers)**: BERT uses a Transformer architecture to generate contextual embeddings for sentences. It takes into account the context on both sides of a word, making it powerful for tasks like question answering and text classification.
- **ELMo (Embeddings from Language Models)**: ELMo uses bi-directional LSTMs to generate word representations that change depending on the context, enabling nuanced text understanding.

Text embeddings can be used to analyze the coherence and structure of a paragraph. AI-generated text may lack natural flow, resulting in embeddings that deviate from typical patterns found in human writing. For example, embeddings generated by BERT can help detect whether the logical flow of sentences is consistent with human writing styles.

**Image Embeddings** Image embeddings capture visual features and semantic information from images, allowing them to be represented as vectors in a continuous space. These embeddings are typically generated using deep learning models like Convolutional Neural Networks (CNNs). Popular models include:

- **VGG (Visual Geometry Group)**: Uses a deep CNN to extract rich features from images, useful for tasks like image classification and object detection.
- **ResNet (Residual Networks)**: Introduces skip connections to handle deeper networks, which improves performance on complex image datasets.
- **Inception (GoogLeNet)**: Uses multiple convolution filter sizes within the same layer to capture different levels of detail in images.
- **EfficientNet**: Balances model complexity and accuracy by scaling depth, width, and resolution efficiently.

### Use Case: Cross-Modal Detection

While not directly related to text, concepts from image embeddings can inspire techniques for detecting AI-generated text. For instance, similar to identifying anomalies in visual patterns, embeddings can be used to detect unnatural patterns in writing.

**Audio Embeddings** Audio embeddings transform audio signals into dense vectors that capture the relevant features and characteristics of sound. These embeddings are typically generated using deep neural networks like CNNs, RNNs, or hybrid models combining both.

### Example Models

- **Wave2Vec**: A model for generating embeddings from raw audio waveforms.

- **Speech2Vec**: Captures relationships between spoken words similar to how Word2Vec captures text relationships.

**Use Case: Transcribed Text Analysis** In the context of detecting AI-generated text, audio embeddings can be used to analyze transcripts. By converting spoken language into text and generating embeddings, patterns in AI-generated audio scripts can be identified.

**Graph Embeddings** Graph embeddings represent nodes, edges, or entire subgraphs in a continuous vector space. They are commonly used to capture the structure and relationships within graph-based data, such as social networks, recommendation systems, and biological networks. Techniques include:

- **Node2Vec**: Generates embeddings for nodes in a graph by simulating random walks.
- **Graph Convolutional Networks (GCNs)**: Extend the idea of CNNs to graph-structured data, allowing the extraction of features from non-Euclidean spaces.
- **DeepWalk**: Uses random walks to learn embeddings that capture network structure.

**Use Case: Detecting AI-Generated Text in Hyperlinked Documents** Graph embeddings can be used to analyze relationships between different subsections of a document or between linked web pages. For example, AI-generated content may have unnatural linking patterns, which can be detected using graph embeddings.

#### 7.11.4 How Embeddings Are Created

The process of creating embeddings involves multiple steps:

1. **Choose or Train an Embedding Model**: Select an appropriate model based on the type of data. For text, models like Word2Vec, GloVe, or BERT are popular; for images, CNN-based models like ResNet are commonly used.
2. **Prepare Data**: Preprocess the data to ensure it is suitable for the chosen model. For text, this involves tokenization, stopword removal, and normalization. For images, resizing and normalization may be required.
3. **Load or Train the Model**: If using a pre-trained model, load its weights and architecture. For custom tasks, training may involve feeding the data into the model and fine-tuning its parameters.
4. **Generate Embeddings**: Convert each data point (text, image, audio, or graph) into a dense vector representation.
5. **Integrate Embeddings**: Use the generated embeddings for downstream applications such as clustering, similarity search, classification, or anomaly detection.

## Real-World Examples of Embeddings

### 1. Natural Language Processing (NLP)

- **Sentiment Analysis:** Word embeddings like Word2Vec or GloVe are used to understand the sentiment conveyed in a text.
- **Question Answering:** BERT embeddings enable the extraction of relevant information from large text corpora.
- **Document Similarity:** Doc2Vec can be used to find semantically similar documents, aiding in plagiarism detection.

### 2. Computer Vision

- **Image Classification:** CNNs like VGG and ResNet extract image features for classification tasks.
- **Image Retrieval:** CLIP models enable cross-modal search by learning joint embeddings for text and images.
- **Facial Recognition:** FaceNet embeddings measure facial similarity for identity verification.

### 3. Recommender Systems

- **Collaborative Filtering:** Embeddings are used to match users with items they are likely to prefer.
- **Product Recommendations:** Word embeddings can analyze product descriptions to recommend similar items.

### 4. Cross-Modal Applications

- **Multimodal Translation:** MUSE can translate text between languages and connect text with images.
- **Cross-Modal Search:** Joint embeddings for different modalities (text, image) enable efficient cross-modal retrieval.

### 5. Anomaly Detection

- **Network Anomalies:** Graph embeddings help detect unusual patterns in network traffic.
- **Fraud Detection:** Transaction embeddings identify patterns in financial data that may indicate fraud.

## 7.12 One-Hot Encoding for Word Embeddings

One-hot encoding is a simple and widely used method for representing words as vectors in a high-dimensional space. **The core idea is to assign a unique index to each word in the vocabulary, and then represent each word as a vector with the corresponding index set to 1, while all other indices are set to 0.** This approach is effective for converting discrete words into numerical representations, which can then be used in machine learning models.

### 7.12.1 Definition of One-Hot Vector for Words

Suppose we have a vocabulary consisting of  $n$  words, such as  $\{\text{anh}, \text{em}, \text{gia đình}, \text{bạn bè}, \dots\}$ . When each word is mapped to its index, we can define a one-hot vector for each word. The one-hot vector of the  $i$ -th word, where  $i \leq n - 1$ , is represented as:

$$e_i = [0, 0, \dots, 1, \dots, 0] \in \mathbb{R}^n$$

Here, the  $i$ -th element of the vector is 1, and all other elements are 0. For example: - The word "anh" might have index 0, so its one-hot vector is  $[1, 0, 0, 0, \dots]$ . - The word "gia đình" might have index 2, so its one-hot vector is  $[0, 0, 1, 0, \dots]$ .

This encoding method assigns a unique vector to each word, ensuring that each word is represented by a sparse vector where only one element is non-zero.

### 7.12.2 One-Hot Encoding Process

In order to apply one-hot encoding in a practical scenario, we first need to build the vocabulary and assign indices to each word. Once we have the index of each word, we can encode it into a one-hot vector.

For instance, the word *anh* has index 0, so its one-hot vector will be  $[1, 0, 0, 0, \dots]$ , while the word *gia đình* with index 2 will be represented as  $[0, 0, 1, 0, \dots]$ .

### 7.12.3 One-Hot Encoding in Python with Sklearn

In Python, we can implement One-Hot Encoding using the `OneHotEncoder` class from the `sklearn.preprocessing` module. However, before encoding, we must first assign indices to the words using `LabelEncoder`.

The following Python code demonstrates the process of transforming words into one-hot encoded vectors:

Listing 7.2: One-hot Encoding

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

```

2 import numpy as np

# Define the words and encode them
3 words = ['anh', 'em', 'gia dinh', 'ban be', 'anh', 'em']
4 le = LabelEncoder()
5 le.fit(words)

# Print class of words and their number transformation
6 print('Class of words: ', le.classes_)
7 x = le.transform(words)
8 print('Convert to number: ', x)

# Inverse transformation to categories
9 print('Invert into classes: ', le.inverse_transform(x))

# Perform One-Hot Encoding
10 oh = OneHotEncoder()
11 classes_indices = list(zip(le.classes_, np.arange(len(le.classes_))))
12 oh.fit(classes_indices)
13 print('One-hot categories and indices:', oh.categories_)

# Convert words into one-hot encoded form
14 words_indices = list(zip(words, x))
15 one_hot = oh.transform(words_indices).toarray()
16 print('Transform words into one-hot matrices: \n', one_hot)

# Inverse transform to categories from one-hot matrices
17 print('Inverse transform to categories from one-hot matrices: \n', oh.
18     inverse_transform(one_hot))

```

### Explanation of Python Code

- First, the words are encoded using `LabelEncoder`, which assigns a unique index to each word.
- Then, we use `OneHotEncoder` to convert the indexed words into one-hot encoded vectors.
- The `fit()` function of `OneHotEncoder` trains the model on the word indices, and `transform()` generates the corresponding one-hot encoded vectors.
- The resulting one-hot matrix is a sparse representation where each word is represented by a vector containing a single 1 at the position corresponding to its index, with all other elements set to 0.

#### 7.12.4 Conclusion

One-hot encoding is an effective technique for representing words in a vector space, particularly when working with discrete sets of categorical data. However, it has limitations, including high dimensionality for large vocabularies and a lack of semantic meaning between words. Despite these limitations, one-hot encoding remains a foundational method in many natural language processing applications, particularly for simple tasks like word classification and clustering.

**Note:** One-hot encoding does not capture the semantic similarity between words, meaning that words with similar meanings (e.g., *man* and *woman*) will not have similar vector representations. This is one reason why more advanced techniques like word embeddings (e.g., Word2Vec, GloVe) are often preferred for tasks that require capturing word relationships.

## 7.13 Pre-trained Word Embeddings: Word2Vec and GloVe

### 7.13.1 Introduction

Pre-trained word embeddings are a key technique in natural language processing (NLP) that transform words into dense vector representations in a continuous vector space. These embeddings capture semantic relationships between words, enabling models to understand and process language more effectively. Two prominent approaches are **Word2Vec** and **GloVe**, which we will explore in detail.

### 7.13.2 Word2Vec

#### 7.13.2.1 History

In 2010, Tomás Mikolov (then at Brno University of Technology) with co-authors applied a simple recurrent neural network with a single hidden layer to language modelling.

Word2Vec was created, patented, and published in 2013 by a team of researchers led by Mikolov at Google. The original paper was initially rejected by the ICLR conference. It also took months for the code to be approved for open-sourcing. Other researchers helped analyze and explain the algorithm.

Embedding vectors created using the Word2Vec algorithm have advantages over earlier algorithms such as n-grams and latent semantic analysis. GloVe, developed at Stanford, was designed as a competitor. Mikolov argued that fastText outperformed GloVe when trained on the same dataset.

As of 2022, Transformer-based models, such as ELMo and BERT, which build on Word2Vec by adding attention layers, are regarded as state-of-the-art in NLP.

#### Overview

The core idea behind Word2Vec can be summarized as follows:

- Words that appear in similar contexts tend to have similar meanings.
- It is possible to predict a word based on its surrounding context. For instance, given the incomplete sentence "Hà Nội là ... của Việt Nam" (Hanoi is the ... of Vietnam), a high-probability word to fill in the blank would be "thủ đô" (capital). By processing a complete sentence like "Hà Nội là thủ đô của Việt Nam" (Hanoi is the capital of Vietnam), Word2Vec learns to generate embeddings such that the probability of predicting "thủ đô" given its context is maximized.

#### How Word2Vec Works

Word2Vec represents words as high-dimensional vectors that capture the semantic and syntactic relationships between words. The model estimates these representations by analyzing large amounts of text data. The vectors are constructed in such a way that words appearing in similar contexts are placed closer together in the vector space, typically measured using cosine similarity.

For example, words like "walk" and "ran" or "but" and "however" are located near each other in the vector space due to their similar contextual usage. Similarly, place names like "Berlin" and "Germany" are also closely aligned in vector space due to their contextual relationship.

## Applications Beyond NLP

Although Word2Vec was originally developed for NLP tasks, the underlying principles can be extended to other domains. For example, the same concept can be applied to build models like `product2vec`, which generate embeddings for products such as food items and household goods. By leveraging co-occurrence patterns in transaction data, `product2vec` can identify products that are frequently purchased together and represent them in a continuous vector space.

In summary, Word2Vec is a powerful tool for learning vector representations of words, enabling models to perform tasks such as detecting synonyms, understanding analogies, and suggesting words to complete partial sentences. Its flexibility has led to its application in various fields beyond language processing, making it a versatile tool in the realm of data science.

### 7.13.2.2 CBOW (Continuous Bag of Words)

A corpus  $C$  is a sequence of words. Both Continuous Bag of Words (CBOW) and Skip-gram are methods used to learn word embeddings, generating one vector per word appearing in the corpus. Let  $V$  be the set of all words appearing in the corpus  $C$ . Our goal is to learn one vector  $\mathbf{v}_w \in \mathbb{R}^n$  for each word  $w \in V$ .

The idea of the Skip-gram model is that the vector of a word should be close to the vectors of each of its neighboring words. On the other hand, the CBOW model aims to make the vector-sum of a word's neighbors close to the vector of the target word.

In the original publication, "closeness" is measured using the softmax function, but the framework allows for other measures of closeness.

**Continuous Bag of Words (CBOW) Model** The CBOW model predicts a target word given its surrounding context words. Suppose we want each word in the corpus to be predicted using every other word within a context window of size 4. The set of context words is defined as:

$$N = \{-4, -3, -2, -1, +1, +2, +3, +4\}.$$

The training objective of the CBOW model is to maximize the probability of predicting the target word  $w_i$  using the context words  $\{w_j : j \in N + i\}$ . The objective can be expressed as:

$$\prod_{i \in C} \Pr(w_i | w_j : j \in N + i), \quad (7.10)$$

where  $C$  is the set of all words in the corpus.

Since products can be numerically unstable, we convert the product into a summation using logarithms:

$$\sum_{i \in C} \log \Pr(w_i | w_j : j \in N + i). \quad (7.11)$$

In the CBOW model, our objective is to maximize the **log-probability** of the corpus. The probability model works as follows:

Given the context words  $\{w_j : j \in N + i\}$ , we first compute their vector sum:

$$v = \sum_{j \in N+i} \mathbf{v}_{w_j}, \quad (7.12)$$

where:

- $N$  is the set of indices representing the context window around the target word.
- $\mathbf{v}_{w_j}$  represents the vector of each context word.

### Maximizing the Log-Probability of the Corpus

Next, we calculate the probability of the target word  $w_i$  using a **dot-product-softmax** with every other vector sum (this step is similar to the attention mechanism in Transformers), to obtain the probability:

$$\Pr(w_i | w_j : j \in N + i) := \frac{e^{\mathbf{v}_{w_i} \cdot v}}{\sum_{w \in V} e^{\mathbf{v}_w \cdot v}}, \quad (7.13)$$

where:

- $v = \sum_{j \in N+i} \mathbf{v}_{w_j}$  is the vector sum of context words.
- $\mathbf{v}_{w_i}$  is the vector representation of the target word.
- $V$  is the vocabulary size.

The quantity to be maximized is then expressed as:

$$\sum_{i \in C, j \in N+i} \left( \mathbf{v}_{w_i} \cdot \mathbf{v}_{w_j} - \ln \sum_{w \in V} e^{\mathbf{v}_w \cdot \mathbf{v}_{w_j}} \right). \quad (7.14)$$

### Explanation of Complexity

The term on the left-hand side  $\mathbf{v}_{w_i} \cdot \mathbf{v}_{w_j}$  is efficient to compute. However, the term on the right  $\ln \sum_{w \in V} e^{\mathbf{v}_w \cdot \mathbf{v}_{w_j}}$  is computationally expensive as it involves summing over the entire vocabulary  $V$ .

To optimize this, we use techniques like:

- **Negative Sampling:** Reduces the computational cost by updating only a subset of negative samples.
- **Hierarchical Softmax:** Reduces the complexity to  $O(\log V)$ .

These approximation techniques significantly reduce the computational cost, making it possible to train models on large text corpora.

In summary, the CBOW model uses a **context window to predict the target word by maximizing the log-probability of its occurrence**. The model leverages numerical approximation methods like negative sampling and hierarchical softmax to optimize training efficiency.

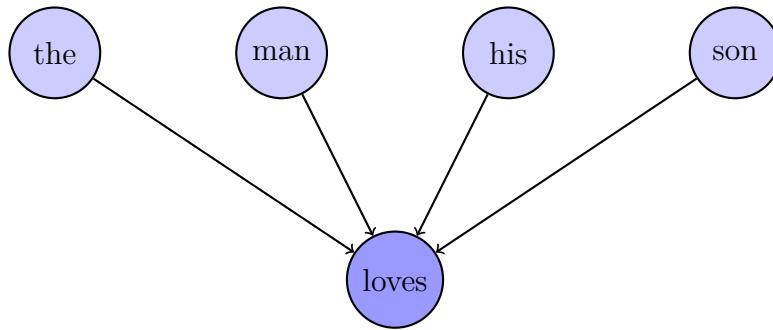


Figure 7.11: Continuous Bag of Words (CBOW) Model: The context words ("the", "man", "his", "son") are used to predict the target word "loves".

### 7.13.2.3 Skip-gram Model

#### Introduction

The Skip-gram model is designed to **predict context words given a target word in the center of a context window**. Let  $V$  be the set of all words in the vocabulary. The goal is to learn a vector representation  $\mathbf{v}_w \in \mathbb{R}^n$  for each word  $w \in V$ .

In the Skip-gram model, the input is the center word, and the predictions are the context words. For example, if we have a word array  $W$ , and  $W(i)$  is the input (center word), then the context words are  $W(i - 2), W(i - 1), W(i + 1), W(i + 2)$  when using a sliding window of size 2.

Let's define some variables:

- $V$ : Number of unique words in our corpus (Vocabulary)
- $x$ : Input layer (One-hot encoding of the input word)
- $N$ : Number of neurons in the hidden layer of the neural network
- $W$ : Weights between input layer and hidden layer

- $W'$ : Weights between hidden layer and output layer
- $y$ : Softmax output layer having probabilities for every word in our vocabulary

## Training Objective

Given a target word  $w_t$  and its context words  $\{w_c\}$ , the objective is to maximize the probability of predicting the context words given the target word:

Consider the example where the target word is “fox” and the context words are “quick,” “brown,” “jumps,” and “over.” The task of predicting the probability of the context words given the target word is modeled by:

$$\mathbb{P}(\text{"quick"}, \text{"brown"}, \text{"jumps"}, \text{"over"} \mid \text{"fox"})$$

We can assume that the occurrence of a context word, given the target word, is independent of the other context words. Therefore, we approximate the above probability as:

$$\mathbb{P}(\text{"quick"} \mid \text{"fox"}) \cdot \mathbb{P}(\text{"brown"} \mid \text{"fox"}) \cdot \mathbb{P}(\text{"jumps"} \mid \text{"fox"}) \cdot \mathbb{P}(\text{"over"} \mid \text{"fox"})$$

Source Text	Training Samples
The quick brown fox jumps over the lazy dog. →	(the, quick) (the, brown)
The quick brown fox jumps over the lazy dog. →	(quick, the) (quick, brown) (quick, fox)
The quick brown fox jumps over the lazy dog. →	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The quick brown fox jumps over the lazy dog. →	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

Figure 7.12: Skip-Gram Model Example: Demonstrating the generation of training samples by pairing context words with a target word from the source text.

This simplification assumes that each context word is conditionally independent of the others when the target word is known, which is a common assumption in many probabilistic models like **Skip-Gram**.

To simplify, we assume the context words are conditionally independent given the target word:

$$\prod_{c \in C_t} \mathbb{P}(w_c | w_t)$$

**Note:** Assuming the context words appear independently of each other around the target word contradicts the idea of Word2Vec, where words in the same context are related to each other. However, this assumption simplifies the model and reduces the complexity while still yielding reasonable results.

Let's assume the target word has an index  $t$  in the vocabulary  $V$ , and the set of indices for the corresponding context words is  $C_t$ . The size of  $C_t$  varies from  $C/2$  (if  $w_t$  is at the beginning or end of the sentence) to  $C$  (if  $w_t$  is in the middle of the sentence and there are enough context words on each side).

To avoid computational errors when multiplying small numbers less than 1, this optimization problem is typically transformed into a minimization problem using the logarithm (often referred to as negative log loss):

$$-\sum_{c \in C_t} \log \mathbb{P}(w_c | w_t)$$

The conditional probability  $\mathbb{P}(w_c | w_t)$  is defined by:

$$\mathbb{P}(w_c | w_t) = \frac{e^{\mathbf{u}_t^T \mathbf{v}_c}}{\sum_{i=1}^N e^{\mathbf{u}_t^T \mathbf{v}_i}}$$

where  $N$  is the size of the vocabulary  $V$ . Here,  $e^{\mathbf{u}_t^T \mathbf{v}_c}$  represents the relationship between the target word  $w_t$  and the context word  $w_c$ . The higher this value, the larger the probability. The dot product  $\mathbf{u}_t^T \mathbf{v}_c$  also represents the similarity between the two vectors.

This expression is very similar to the Softmax function. Defining the probability as in equation (1) ensures that:

$$\sum_{w \in V} \mathbb{P}(w | w_t) = 1$$

In summary, the loss function corresponding to the target word  $w_t$  with respect to  $U$  and  $V$  is given by:

$$\mathcal{L}(\mathbf{U}, \mathbf{V}; w_t) = -\sum_{c \in C_t} \log \frac{e^{\mathbf{u}_t^T \mathbf{v}_c}}{\sum_{i=1}^N e^{\mathbf{u}_t^T \mathbf{v}_i}}$$

### Skip-gram Model in Neural Network Form Explanation

The Skip-gram Word2Vec model can be illustrated as a simple neural network with a single hidden layer. In this architecture:

- The input layer takes a one-hot encoded vector representing the target word  $w_t$ .
- The hidden layer performs a matrix multiplication with the weight matrix  $U$  to produce the hidden layer output  $\mathbf{u}_t$ .

- The output layer then multiplies the hidden layer output with another weight matrix  $V$  to produce  $\mathbf{u}_t^\top V$ , which represents the logits before applying the softmax activation function.

This simple architecture enables Word2Vec to perform well even with a large vocabulary (potentially containing millions of words). The input and output layers of this neural network have dimensions equal to the size of the vocabulary.

### Illustration of the Skip-gram Model

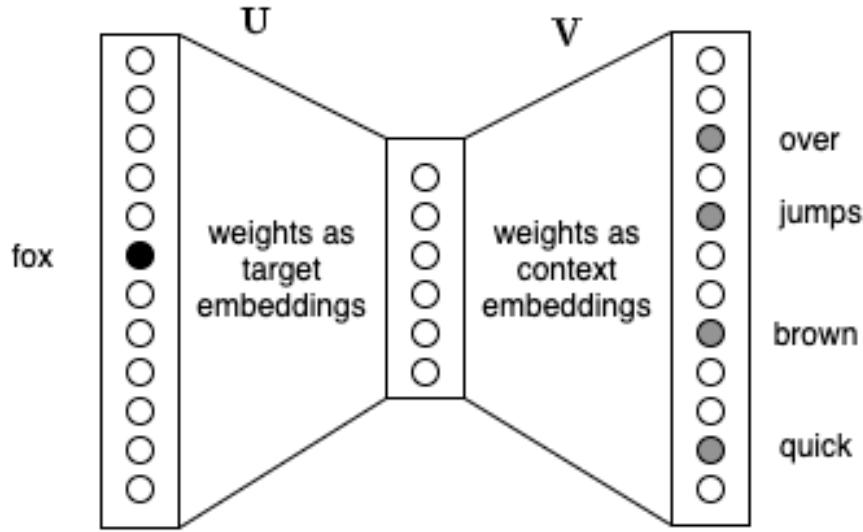


Figure 7.13: Word2Vec: Skip-Gram

### Optimizing the Loss Function

The optimization of the two weight matrices  $U$  and  $V$  is carried out using gradient descent algorithms. These optimization algorithms require the computation of gradients for each matrix.

Consider the term:

$$\log \mathbb{P}(w_c | w_t) = \log \left( \frac{e^{\mathbf{u}_t^\top \mathbf{v}_c}}{\sum_{i=1}^N e^{\mathbf{u}_t^\top \mathbf{v}_i}} \right) = \mathbf{u}_t^\top \mathbf{v}_c - \log \left( \sum_{i=1}^N e^{\mathbf{u}_t^\top \mathbf{v}_i} \right)$$

The gradient with respect to  $u_t$  is:

$$\frac{\partial \log \mathbb{P}(w_c | w_t)}{\partial \mathbf{u}_t} = \mathbf{v}_c - \sum_{j=1}^N \frac{e^{\mathbf{u}_t^\top \mathbf{v}_j} \mathbf{v}_j}{\sum_{i=1}^N e^{\mathbf{u}_t^\top \mathbf{v}_i}} = \mathbf{v}_c - \sum_{j=1}^N \mathbb{P}(w_j | w_t) \mathbf{v}_j$$

Thus, although this gradient is quite elegant, we still need to compute the probabilities  $\mathbb{P}(w_j | w_t)$ . Each of these probabilities depends on the entire weight matrix  $V$  and the vector

$\mathbf{u}_t$ . Therefore, we need to update a total of  $N \times d + d$  weights, which is clearly a very large number when  $N$  is large.

### Approximation of the Loss Function and Negative Sampling

To avoid updating such a large number of parameters in one iteration, an approximation method has been proposed that significantly improves computational speed. Each probability  $\mathbb{P}(w_c|w_t)$  is modeled by a sigmoid function instead of the softmax function:

$$\mathbb{P}(w_c|w_t) = \frac{1}{1 + e^{\mathbf{u}_t^\top \mathbf{v}_c}}$$

Note that the sum of the probabilities  $\sum_{w_c \in V} \mathbb{P}(w_c|w_t)$  is no longer equal to 1. However, it still represents the probability of the presence of the context word  $w_c$  alongside the target word  $w_t$ .

At this point, calculating  $\mathbb{P}(w_c|w_t)$  depends only on the vector  $\mathbf{u}_t$  and the vector  $\mathbf{v}_c$  (instead of the entire matrix  $V$ ). Corresponding to this term, only  $2d$  weights need to be updated for each pair  $(w_t, w_c)$ . This number of weights does not depend on the size of the vocabulary, making this model feasible even with a very large  $N$ .

However, there is a major issue with this modeling approach!

Since there is no constraint between the probabilities  $\mathbb{P}(w_c|w_t)$ , trying to maximize each probability will lead to a situation where all the  $\mathbb{P}(w_c|w_t)$  are high. This is achieved when  $e^{\mathbf{u}_t^\top \mathbf{v}_c}$  approaches 0. All the elements of  $U$  and  $V$  would need to approach infinity, satisfying this condition. This approximation now becomes trivial and meaningless. To avoid this issue, we need to introduce additional constraints, so that there are other probabilities  $\mathbb{P}(w_n|w_t)$  that must be minimized when considering the target word  $w_t$ .

The essence of the original optimization problem is to build a model such that for each target word, the probability of a context word occurring is high, while the probabilities of all the other words outside of the context are low — this is represented in the softmax function. To limit the computation, in this approach, we randomly sample a few words outside the context to optimize. The words in the context are called "positive words", and the words outside the context are called "negative words". Therefore, this method is also known as "negative sampling".

Thus, for each target word, we have a set of context words with labels 1 and 0 corresponding to the context words (called positive context) and the negative context words sampled from outside the positive context set. For the positive context words,  $-\log(\mathbb{P}(w_c|w_t))$  is similar to the loss function in logistic regression with labels equal to 1. Similarly, we can use  $-\log(1 - \mathbb{P}(w_c|w_t))$  as the loss function for the negative context words with labels equal to 0.

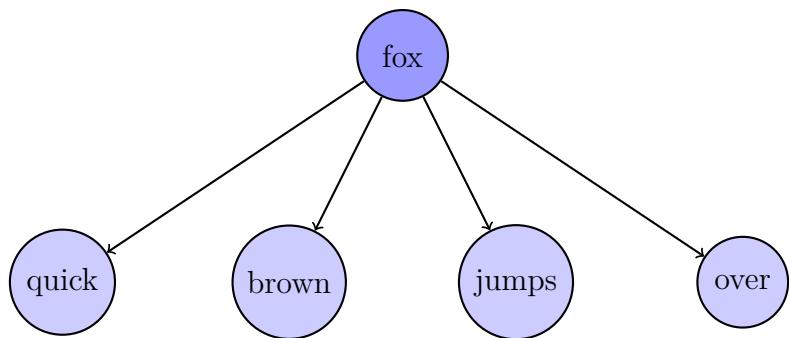


Figure 7.14: Skip-gram Model: The target word "fox" is used to predict context words ("quick", "brown", "jumps", "over").

#### 7.13.2.4 Parameterization of Word2Vec

##### Training Algorithm

A Word2Vec model can be trained using either **hierarchical softmax** or **negative sampling**. The hierarchical softmax method uses a Huffman tree to reduce calculations. In contrast, negative sampling minimizes the log-likelihood of sampled negative instances:

$$\mathcal{L} = - \sum_{c \in C_t} \log \left( \frac{e^{\mathbf{u}_t^\top \mathbf{v}_c}}{\sum_{i=1}^N e^{\mathbf{u}_t^\top \mathbf{v}_i}} \right)$$

Hierarchical softmax is more effective for infrequent words, while negative sampling works better for frequent words.

##### Sub-sampling

To speed up training, high-frequency and low-frequency words can be sub-sampled. Words with a frequency above or below a threshold may be removed.

##### Dimensionality

Increasing the dimensionality of embeddings improves quality, but gains diminish after a certain point. The typical range is between 100 and 1,000 dimensions.

##### Context Window

The size of the context window determines how many words before and after a given word are included as context. Recommended values:

- Skip-gram: 10
- CBOW: 5

#### 7.13.2.5 Extensions to Word2Vec

##### doc2vec

The **doc2vec** model generates distributed representations of sentences, paragraphs, or entire documents. It extends Word2Vec with two architectures:

- **Distributed Memory Model of Paragraph Vectors (PV-DM)**: Similar to CBOW but includes a document identifier.
- **Distributed Bag of Words version of Paragraph Vector (PV-DBOW)**: Similar to Skip-gram but uses paragraph identifiers instead of words.

##### top2vec

The **top2vec** model uses document embeddings to estimate topic distributions. It uses **UMAP** for dimensionality reduction and **HDBSCAN** for clustering.

## **BioVectors**

**BioVectors** extend Word2Vec for biological sequences, such as DNA, RNA, and proteins. Examples include:

- **ProtVec** for proteins.
- **GeneVec** for gene sequences.

## **Intelligent Word Embeddings (IWE)**

IWE models combine Word2Vec with semantic dictionary mapping to handle out-of-vocabulary words in domains like medicine.

### 7.13.3 GloVe (Global Vectors for Word Representation)

GloVe (Global Vectors for Word Representation), developed by Stanford University, is an **unsupervised learning algorithm designed to generate distributed word embeddings**. Unlike Word2Vec, which primarily focuses on local context windows, GloVe captures **global word-word co-occurrence statistics** from a corpus. The model works by mapping words into a meaningful space where the distance between words reflects their semantic similarity. By combining features from both global matrix factorization and local context window methods, GloVe produces word representations that exhibit interesting linear substructures in the word vector space.

Launched in 2014 as an open-source project, GloVe was developed as a competitor to Word2Vec. The original paper highlighted several improvements of GloVe over Word2Vec, particularly its ability to leverage global statistics more effectively. Since then, however, both Word2Vec and GloVe have been overshadowed by Transformer-based models like ELMo and BERT, which incorporate multiple neural network attention layers on top of word embeddings, pushing the boundaries of NLP. Despite this, GloVe remains an important milestone in the evolution of word representation models.

*You shall know a word by the company it keeps* (Firth, J. R. 1957:11)

The idea of GloVe is to construct, for each word  $i$ , two vectors  $w_i$  and  $\tilde{w}_i$ , such that the relative positions of the vectors capture part of the statistical regularities of the word  $i$ . The statistical regularity is defined as the co-occurrence probabilities. Words that resemble each other in meaning should also resemble each other in co-occurrence probabilities.

#### 7.13.3.1 Word Counting

Let the vocabulary be  $V$ , the set of all possible words (aka "tokens"). Punctuation is either ignored, or treated as vocabulary, and similarly for capitalization and other typographical details.

If **two words occur close** to each other, then we say that they **occur in the context of each other**. For example, if the context length is 3, then we say that in the following sentence

GloVe1, coined2 from3 Global4 Vectors5, is6 a7 model8 for9 distributed10 word11 representation12

the word "model8" is in the context of "word11" but not the context of "representation12".

A word is not in the context of itself, so "model8" is not in the context of the word "model8", although, if a word appears again in the same context, then it does count.

Let  $X_{ij}$  be the number of times that the word  $j$  appears in the context of the word  $i$  over the entire corpus.

**Example** If the corpus is just "I don't think that that is a problem", we have:

$$X_{\text{that}, \text{that}} = 2$$

since the first "that" appears in the second one's context, and vice versa.

Let  $X_i = \sum_{j \in V} X_{ij}$  be the number of words in the context of all instances of word  $i$ . By counting, we have:

$$X_i = 2 \times (\text{context size}) \times \#(\text{occurrences of word } i)$$

except for words occurring right at the start and end of the corpus.

### 7.13.3.2 Probabilistic Modelling

Let

$$\mathbf{P}_{ik} := \mathbb{P}(k|i) := \frac{X_{ik}}{X_i}$$

be the **co-occurrence probability**. That is, if one samples a random occurrence of the word  $i$  in the entire document, and a random word within its context, that word is  $k$  with probability  $\mathbf{P}_{ik}$ . Note that  $\mathbf{P}_{ik} \neq \mathbf{P}_{ki}$  in general.

For example, in a typical modern English corpus,  $\mathbf{P}_{\text{ado,much}}$  is close to one, but  $\mathbf{P}_{\text{much,ado}}$  is close to zero. This is because the word "ado" is almost only used in the context of the archaic phrase "much ado about", but the word "much" occurs in all kinds of contexts.

For example, in a 6 billion token corpus, we have:

Probability and Ratio	$k = \text{solid}$	$k = \text{gas}$	$k = \text{water}$	$k = \text{fashion}$
$\mathbb{P}(k   \text{ice})$	$1.9 \times 10^{-4}$	$6.6 \times 10^{-5}$	$3.0 \times 10^{-3}$	$1.7 \times 10^{-5}$
$\mathbb{P}(k   \text{steam})$	$2.2 \times 10^{-5}$	$7.8 \times 10^{-4}$	$2.2 \times 10^{-3}$	$1.8 \times 10^{-5}$
$\mathbb{P}(k   \text{ice}) / \mathbb{P}(k   \text{steam})$	8.9	$8.5 \times 10^{-2}$	1.36	0.96

Inspecting the table, we see that the words "ice" and "steam" are indistinguishable along the "water" (often co-occurring with both) and "fashion" (rarely co-occurring with either), but distinguishable along the "solid" (co-occurring more with ice) and "gas" (co-occurring more with "steam").

The idea is to learn two vectors  $w_i$  and  $\tilde{w}_i$  for each word  $i$ , such that we have a **multinomial logistic regression**:

$$w_i^T \tilde{w}_j + b_i + \tilde{b}_j \approx \ln \mathbf{P}_{ij}$$

and the terms  $b_i, \tilde{b}_j$  are unimportant parameters.

This means that if the words  $i$  and  $j$  have similar co-occurrence probabilities  $(\mathbf{P}_{ik})_{k \in V} \approx (\mathbf{P}_{jk})_{k \in V}$ , then their vectors should also be similar:  $w_i \approx w_j$ .

The equation and the surrounding text describe the core idea of GloVe's multinomial logistic regression model. It represents the goal of the GloVe model, which is to learn the word vectors  $w_i$  and  $\tilde{w}_j$  for each word  $i$  and its context word  $j$  such that the dot product of their vectors, plus the bias terms, approximates the logarithm of the co-occurrence probability  $\mathbf{P}_{ij}$ . Below is a detailed explanation of the parameters:

1.  $w_i$  and  $\tilde{w}_j$ : These are the word vectors representing the **target word** and the **context word**. The goal is to learn these vectors so that words with similar meanings (and thus similar co-occurrence patterns) are represented by similar vectors.
2.  $b_i$  and  $\tilde{b}_j$ : These are the **bias terms** for the target word  $i$  and the context word  $j$ , which help adjust the model to account for the varying frequency of words in the corpus.
3.  $\mathbf{P}_{ij}$ : This is the **co-occurrence probability of word**  $j$  occurring in the context of word  $i$ . It is calculated based on the frequency with which word  $j$  appears in the context of word  $i$  in the corpus.
4.  $\ln \mathbf{P}_{ij}$ : The **logarithm of the co-occurrence probability** is used to transform the probabilities into a more manageable scale for optimization.

The key idea is that if two words  $i$  and  $j$  have similar co-occurrence probabilities with other words in the vocabulary, i.e.,  $(\mathbf{P}_{ik})_{k \in V} \approx (\mathbf{P}_{jk})_{k \in V}$ , their corresponding word vectors  $w_i$  and  $w_j$  should also be similar. This ensures that words with similar meanings or contexts are placed near each other in the learned word embedding space. Thus, GloVe effectively captures semantic relationships between words based on their co-occurrence patterns in the corpus.

### 7.13.3.3 Logistic Regression

Naively, logistic regression can be run by **minimizing the squared loss**:

$$L = \sum_{i,j \in V} (w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \ln \mathbf{P}_{ik})^2$$

However, this would be noisy for rare co-occurrences. To fix the issue, the squared loss is weighted so that the loss is slowly ramped-up as the absolute number of co-occurrences  $X_{ij}$  increases:

$$L = \sum_{i,j \in V} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \ln \mathbf{P}_{ik})^2$$

where

$$f(x) = \begin{cases} \left(\frac{x}{x_{\max}}\right)^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}$$

and  $x_{\max}, \alpha$  are hyperparameters. In the original paper, the authors found that  $x_{\max} = 100$ ,  $\alpha = 3/4$  worked well in practice.

#### 7.13.3.4 Use

Once a model is trained, we have 4 trained parameters for each word:  $w_i, \tilde{w}_i, b_i, \tilde{b}_i$ . The parameters  $b_i, \tilde{b}_i$  are irrelevant, and only  $w_i, \tilde{w}_i$  are relevant.

The authors recommended using  $w_i + \tilde{w}_i$  as the final representation vector for word  $i$ , because empirically it worked better than  $w_i$  or  $\tilde{w}_i$  alone.

#### 7.13.3.5 Nearest Neighbors

The Euclidean distance (or cosine similarity) between two word vectors provides an effective method for measuring the linguistic or semantic similarity of the corresponding words. Sometimes, the nearest neighbors according to this metric reveal rare but relevant words that lie outside an average human's vocabulary. For example, here are the closest words to the target word **frog**: *frog, frogs, toad, litoria, leptodactylidae, rana, lizard, eleutherodactylus*.



Figure 7.15: GloVe: Nearest Neighbors Example (Stanford)

**Explanation:** The nearest neighbors of the word *frog* show how GloVe can capture both common and less common words with similar meanings. Some of the words are more scientific or rare terms (e.g., *litoria*, *leptodactylidae*) that are related to *frog* but might not be present in everyday vocabulary. This highlights the ability of GloVe to learn nuanced semantic relationships.

#### 7.13.3.6 Linear Substructures

The similarity metrics used for nearest neighbor evaluations produce a single scalar that quantifies the relatedness of two words. However, this simplicity can be problematic since

two given words almost always exhibit more intricate relationships than can be captured by a single number.

For example, *man* may be regarded as similar to *woman* in that both words describe human beings; on the other hand, the two words are often considered opposites since they highlight a primary axis along which humans differ from one another.

In order to capture in a quantitative way the nuance necessary to distinguish *man* from *woman*, it is necessary for a model to associate more than a single number to the word pair. A natural and simple candidate for an enlarged set of discriminative numbers is the vector difference between the two word vectors. GloVe is designed in such a way that such vector differences capture as much as possible the meaning specified by the juxtaposition of two words.

For example, the following vector differences are useful for capturing the relationship between word pairs:

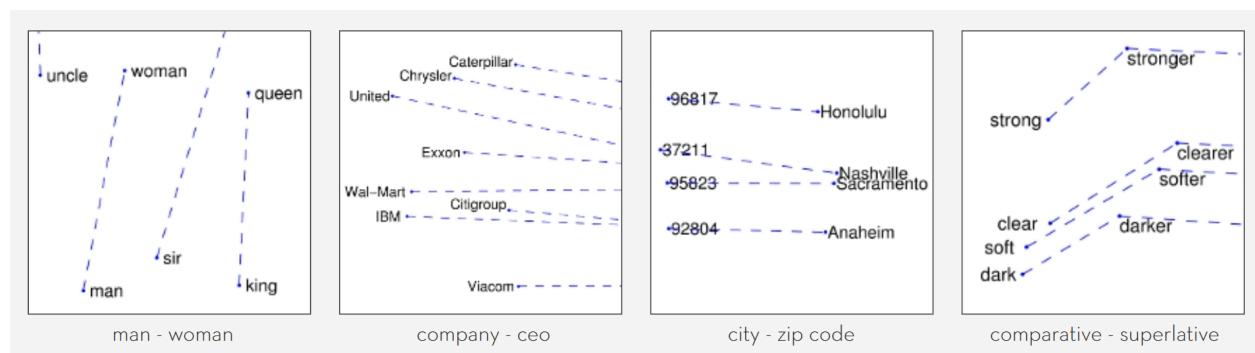


Figure 7.16: GloVe: Linear Structure Example (Stanford)

**Explanation:** GloVe captures subtle relationships between word pairs through vector differences. For example, the vector difference between *man* and *woman* is related to the concept of gender. This approach helps GloVe identify analogies, such as *king* to *queen* or *brother* to *sister*, in a mathematically consistent way.

The underlying concept that distinguishes *man* from *woman*, i.e., sex or gender, may be equivalently specified by various other word pairs, such as *king* and *queen* or *brother* and *sister*. To state this observation mathematically, we might expect that the vector differences  $\text{man} - \text{woman}$ ,  $\text{king} - \text{queen}$ , and  $\text{brother} - \text{sister}$  might all be roughly equal.

#### 7.13.3.7 Training

The GloVe model is trained on the non-zero entries of a global word-word co-occurrence matrix, which tabulates how frequently words co-occur with one another in a given corpus. Populating this matrix requires a single pass through the entire corpus to collect the statistics. For large corpora, this pass can be computationally expensive, but it is a one-time upfront cost. Subsequent training iterations are much faster because the number of non-zero matrix entries is typically much smaller than the total number of words in the corpus.

The tools provided in this package automate the collection and preparation of co-occurrence statistics for input into the model. The core training code is separated from these preprocessing steps and can be executed independently.

## Differences Between Word2Vec and GloVe

- **Context:** Word2Vec uses local context windows, whereas GloVe leverages global co-occurrence statistics across the entire corpus.
- **Training Objective:** Word2Vec predicts context words based on a target word (or vice versa), while GloVe factorizes the co-occurrence matrix to find word vectors.

**Applications of GloVe** GloVe embeddings have been widely used in:

- **Sentiment Analysis:** Understanding sentiment by capturing word associations.
- **Named Entity Recognition (NER):** Identifying entities like names, places, and organizations.
- **Question Answering Systems:** Improving understanding of queries to retrieve relevant information.

#### 7.13.4 Differences in the Properties of Word2Vec and GloVe

The two models, Word2Vec and GloVe, differ significantly in the way they are trained, which leads to subtle differences in the word vectors they generate. GloVe is based on leveraging global word-to-word co-occurrence counts across the entire corpus. In contrast, Word2Vec focuses on co-occurrence within local context windows (i.e., neighboring words).

In practice, however, both models produce similar results for many tasks. Factors such as the dataset on which the models are trained, the length of the vectors, and domain-specific data seem to have a bigger impact on performance than the choice of the model itself. For instance, when using these models for feature extraction in a medical application, significant improvements in performance can be achieved by training the models on a medical-specific dataset. So **how are the Word2Vec and GloVe Models Trained?**

##### 7.13.4.1 How is the Word2Vec Model Trained?

Word2Vec is a feed-forward neural network-based model used to generate word embeddings. There are two commonly used models for training these embeddings:

- **Skip-gram Model:** In this model, each word in the corpus is used as an input, which is sent to a hidden layer (embedding layer), from where it predicts the context words surrounding it. Once the model is trained, the embedding for a specific word is obtained by feeding the word as input and taking the value of the hidden layer as its final embedding vector.
- **CBOW (Continuous Bag of Words) Model:** This model takes the context words surrounding a target word as input, sends them to the hidden layer (embedding layer), and predicts the target word. Once trained, the embedding for a word is obtained by feeding the word as input and taking the value of the hidden layer as the final embedding vector.

The following figure demonstrates these models:

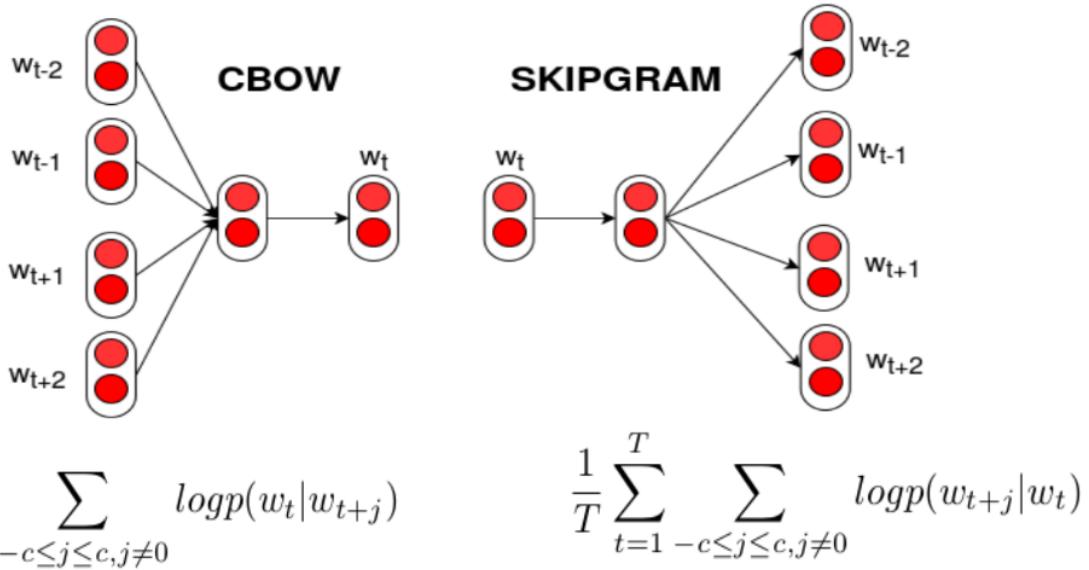


Figure 7.17: CBOW vs Skip-Gram

#### 7.13.4.2 How is the GloVe Model Trained?

GloVe is based on **matrix factorization techniques applied to the word-context co-occurrence matrix**. The process begins by constructing a **large matrix that records the frequency of each word** in a specific context across the corpus. The size of the context matrix is combinatorially large because the number of possible word-context pairs increases rapidly with the size of the vocabulary.

The goal of GloVe is to **factorize this co-occurrence matrix into a lower-dimensional word-feature matrix**, where each row corresponds to a vector representation of a word. This is done by minimizing a "reconstruction loss," which attempts to find low-dimensional representations that can explain most of the variance in the high-dimensional data.

#### 7.13.5 Conclusion

Both Word2Vec and GloVe have significantly advanced the field of NLP by providing methods to capture semantic relationships between words, enabling more sophisticated language understanding in various applications. While Word2Vec focuses on local context, GloVe leverages global co-occurrence statistics to generate embeddings.

## 7.14 Basic Attention

The concept of **attention** is a fundamental mechanism that enables neural network models to **focus selectively on specific parts** of an input sequence while performing tasks such as translation, summarization, and image captioning. Originally introduced to improve machine translation, attention allows models to **dynamically prioritize relevant portions** of an input sequence at each time step, enabling improved handling of long sequences and enhancing performance on various tasks.

### 7.14.1 Mechanics of Basic Attention

In an attention mechanism, the model dynamically computes a set of weights, called **attention scores**, that determine the **relative importance** of each element in an input sequence.

Given an input sequence,  $\mathbf{X} = [x_1, x_2, \dots, x_n]$ , and a query vector  $\mathbf{q}$  representing the model's current focus, the attention mechanism computes a **weighted sum** of the input elements based on their **relevance** to the query. This weighted sum, called the **context vector**, highlights specific parts of the input that are most relevant to the model's task.

The attention score  $e_{ij}$  between a query  $\mathbf{q}_i$  and an input vector  $\mathbf{x}_j$  is typically computed using a **similarity function**, such as **dot product or scaled dot product**, followed by a softmax operation to **normalize** the scores:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^n \exp(e_{ik})}$$

The context vector  $\mathbf{c}_i$  is then computed as a weighted sum of the input vectors:

$$\mathbf{c}_i = \sum_{j=1}^n \alpha_{ij} \mathbf{x}_j$$

This context vector  $\mathbf{c}_i$  represents the parts of the input that are most relevant to the query  $\mathbf{q}_i$ , allowing the model to focus on specific information at each step.

### 7.14.2 Applications of Basic Attention in Sequential Tasks

Attention mechanisms are particularly effective in handling long input sequences where **sequential models** like recurrent neural networks (**RNNs**) may struggle to retain information across **lengthy dependencies**.

By focusing on specific parts of the input, attention allows the model to retain **critical information** without relying solely on sequential memory, making it highly effective for tasks with complex relationships, such as:

- **Machine Translation:** Attention allows translation models to focus on relevant source words for each target word, improving translation quality and handling of longer sentences.
- **Text Summarization:** In summarization, attention enables the model to identify and condense the most important parts of a document.
- **Image Captioning:** Attention helps models focus on relevant regions of an image to generate descriptive captions by treating parts of the image as "words" in a sequence.

### 7.14.3 Attention Variants

Over time, several variants of attention mechanisms have emerged, each tailored to specific types of tasks. **Self-attention**, for example, computes attention within a sequence to capture **intra-sequence dependencies** and is a **core component of transformer models**. These attention variants extend the functionality of basic attention by capturing complex relationships between elements in different ways.

### 7.14.4 Benefits of Basic Attention

The attention mechanism enhances the interpretability of neural network models by providing insights into which parts of the input the model considers important for each output. This interpretability aids in model debugging and helps understand model decisions. Additionally, attention enables efficient parallelization and scalability in transformer architectures, making it instrumental in modern natural language processing systems.

## 7.15 Self-Attention Mechanism: Capturing Contextual Dependencies within Sequences

Self-attention is a specialized mechanism in neural network architectures that **enables each element in an input sequence to attend to all other elements in the sequence**, allowing models to capture **contextual dependencies** effectively. Unlike traditional attention mechanisms, which typically involve an external query, self-attention operates by using **each sequence element as a query, key, and value**, thus capturing the internal structure and relationships within the sequence.

### 7.15.1 Motivation for Self-Attention

Self-attention mechanisms have gained popularity in **sequence transduction tasks** due to their unique ability to handle dependencies across sequences more efficiently than traditional recurrent and convolutional layers. In this section, we **compare the various aspects of self-attention with recurrent and convolutional layers**, particularly focusing on their applications in tasks requiring the transformation of one variable-length sequence of symbol representations  $(x_1, \dots, x_n)$  to another sequence of equal length  $(z_1, \dots, z_n)$ , with  $x_i, z_i \in \mathbb{R}^d$ , such as hidden layers in typical sequence transduction encoders and decoders. The motivation for using self-attention can be understood through **three key considerations**.

#### 7.15.1.1 Computational Complexity Per Layer

One primary factor is the total computational complexity per layer. Unlike **recurrent layers, which require  $O(n)$  sequential operations**, self-attention layers have a lower computational cost when sequence length  $n$  is smaller than the representation dimensionality  $d$ , as is commonly the case with sentence representations in tasks like machine translation. Self-attention layers efficiently process sequences due to their ability to connect all positions in a single forward pass. Additionally, restricting self-attention to a localized neighborhood of size  $r$  around each output position can further optimize performance in tasks involving very long sequences. This neighborhood-based self-attention reduces the maximum path length to  $O(n/r)$ , thus preserving the computational advantage in longer sequences.

#### 7.15.1.2 Parallelization Potential

The second consideration is the degree of parallelization. Self-attention layers are inherently parallelizable, as they allow all positions in the sequence to be processed simultaneously. In contrast, recurrent layers require sequential processing, limiting their potential for parallelization. This distinction makes self-attention mechanisms particularly effective in tasks that benefit from faster computation speeds and large-scale parallel processing.

### 7.15.1.3 Path Length for Long-Range Dependencies

A third, crucial factor is the path length for capturing long-range dependencies. Recurrent neural networks often struggle with long-range dependencies, as the path for information flow in recurrent networks grows linearly with the sequence length. This increased path length can hinder the learning of dependencies in long sequences. Self-attention, on the other hand, provides direct connections between any two positions in the input and output sequences, reducing the path length to a constant. This ability to capture dependencies between distant elements in a sequence facilitates better learning of long-range relationships.

## 7.15.2 Mechanics of Self-Attention

The self-attention process starts with three representations for each word or token in the sequence: the **query**, **key**, and **value**. These are computed by linearly projecting each input vector  $\mathbf{x}_i$  through learned weight matrices:

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i, \quad \mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i, \quad \mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i$$

where  $\mathbf{W}_q$ ,  $\mathbf{W}_k$ , and  $\mathbf{W}_v$  are learned weight matrices. The attention score between each pair of tokens  $i$  and  $j$  is then computed using the dot product of the query and key vectors, scaled by the square root of the dimensionality  $d_k$ :

$$e_{ij} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

The scores are passed through a **softmax function** to obtain normalized attention weights:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^n \exp(e_{ik})}$$

These weights determine the **influence** each token  $j$  has on token  $i$ . The final output of the self-attention mechanism for each token is a weighted sum of all value vectors in the sequence:

$$\mathbf{z}_i = \sum_{j=1}^n \alpha_{ij} \mathbf{v}_j$$

This allows each token to incorporate contextual information from other tokens in the sequence, capturing **long-range dependencies and relationships** that are crucial for understanding complex language and sequential patterns.

### 7.15.3 Applications in Transformer Models

Self-attention is the **foundation of the transformer architecture**, a model that has revolutionized natural language processing by enabling parallelization and scalable training. Transformers use **multi-head self-attention**, where multiple self-attention mechanisms operate in parallel to capture diverse types of dependencies. This architecture has been instrumental in the success of large-scale language models such as BERT and GPT.

### 7.15.4 Benefits of Self-Attention

The self-attention mechanism offers several advantages:

- **Captures Global Context:** By allowing each token to attend to all others, self-attention captures global relationships within the sequence, enhancing the model's contextual understanding.
- **Parallelization:** Unlike recurrent neural networks, self-attention does not require sequential processing, enabling parallelization and reducing computational overhead.
- **Scalability:** The self-attention mechanism can handle large inputs more effectively, making it well-suited for tasks with extensive data requirements.

### 7.15.5 Challenges and Considerations

While self-attention provides powerful context-awareness, it requires quadratic computational complexity relative to the sequence length, which can be computationally demanding for very long sequences. Techniques such as **sparse attention and efficient transformer models** aim to address this limitation by focusing on subsets of sequence elements.

### 7.15.6 Comparison with Convolutional Layers

In contrast, a single convolutional layer with kernel width  $k < n$  does not connect all input and output positions directly. Connecting all positions requires stacking multiple layers, with  $O(n/k)$  layers for contiguous kernels or  $O(\log_k(n))$  for dilated convolutions. This increases the maximum path length between positions, reducing the efficiency in learning dependencies. Furthermore, convolutional layers typically involve higher computational costs than recurrent layers, due to the need for more layers and operations. Using separable convolutions can reduce complexity to  $O(k \cdot n \cdot d + n \cdot d^2)$ , yet this remains comparable to the complexity of a self-attention layer combined with a point-wise feed-forward layer.

## 7.16 Scaled Dot-Product Attention

Scaled dot-product attention is a **key mechanism** in the architecture of **transformer models**, enabling efficient and stable computation of attention scores. It serves as the **mathematical foundation** behind how neural networks focus on specific parts of input sequences, which is particularly useful for **tasks requiring contextual understanding**. In this section, we explain the scaled dot-product attention mechanism, detailing its mathematical formulation, purpose of scaling, and implications for stability in training.

### 7.16.1 Mathematical Formulation of Scaled Dot-Product Attention

Scaled Dot-Product Attention

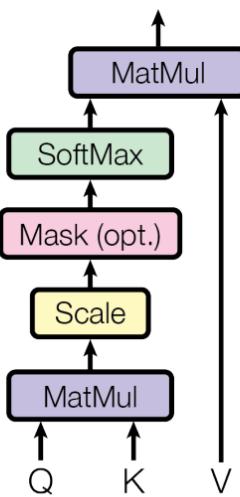


Figure 7.18: Scaled Dot-Product Attention

In the scaled dot-product attention mechanism, each token in a sequence is represented by **three vectors**: *query* ( $\mathbf{q}$ ), *key* ( $\mathbf{k}$ ), and *value* ( $\mathbf{v}$ ), which are computed through linear transformations of the input vectors using learned weight matrices:

$$\mathbf{q} = \mathbf{W}_q \mathbf{x}, \quad \mathbf{k} = \mathbf{W}_k \mathbf{x}, \quad \mathbf{v} = \mathbf{W}_v \mathbf{x}$$

The attention score between a query and each key is calculated using the dot product of  $\mathbf{q}$  and  $\mathbf{k}$ . The score measures the similarity between the query and key, indicating how much focus should be placed on the corresponding value  $\mathbf{v}$ :

$$e = \mathbf{q} \cdot \mathbf{k}$$

To stabilize the attention scores during training, especially for large sequences, the dot product  $e$  is **scaled by the square root of the dimensionality of the key vectors  $d_k$** :

$$e = \frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}}$$

This scaling factor,  $\sqrt{d_k}$ , mitigates the **risk of overly large values in the softmax function** that could otherwise result in **vanishing gradients**, making training unstable.

### 7.16.2 Softmax and Output Computation

After scaling, the scores are passed through a softmax function to obtain the attention weights, which are normalized probabilities representing the importance of each key-value pair for a given query:

$$\alpha = \text{softmax}\left(\frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}}\right)$$

These weights are then used to compute a weighted sum of the values, producing the output of the attention mechanism:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \sum_{i=1}^n \alpha_i \mathbf{v}_i$$

where  $\mathbf{Q}$ ,  $\mathbf{K}$ , and  $\mathbf{V}$  represent the matrices of query, key, and value vectors, respectively. This output vector incorporates information from relevant parts of the input sequence, allowing the model to capture dependencies across tokens effectively.

## 7.17 Multi-Head Attention

Multi-head attention is an **extension of the single-head attention mechanism** that enables models to capture a wider range of patterns in input sequences by applying **multiple attention “heads” in parallel**. Each head independently focuses on different aspects of the input, allowing the model to learn various relationships and contextual dependencies. This section will delve into the mathematical workings of multi-head attention, its advantages, and its significance in transformer architectures.

### 7.17.1 Mechanism of Multi-Head Attention

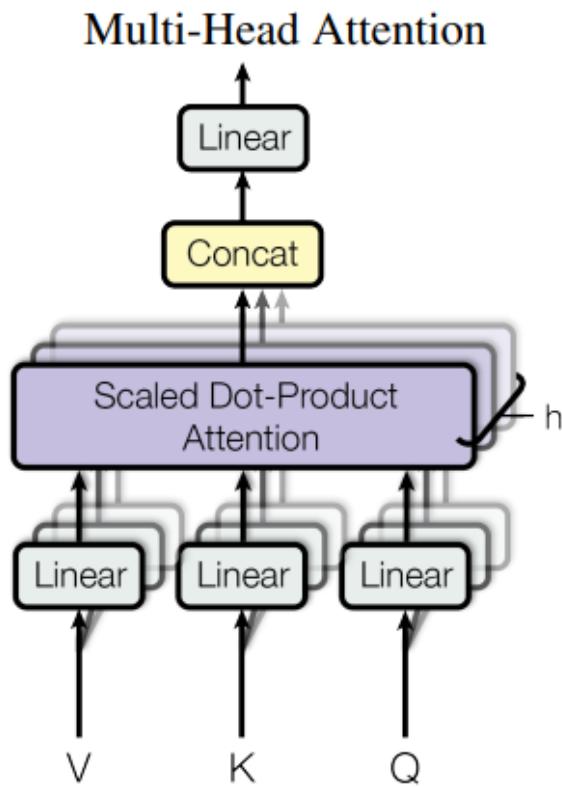


Figure 7.19: Multi-Head Attention consists of several attention layers running in parallel

In multi-head attention, multiple sets of attention mechanisms (or heads) operate on different linear transformations of the input data. Each attention head is initialized with its own learned parameters and performs scaled dot-product attention. Given an input sequence represented as vectors, each head computes its set of query ( $\mathbf{Q}_i$ ), key ( $\mathbf{K}_i$ ), and value ( $\mathbf{V}_i$ ) matrices as follows:

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^Q, \quad \mathbf{K}_i = \mathbf{X}\mathbf{W}_i^K, \quad \mathbf{V}_i = \mathbf{X}\mathbf{W}_i^V$$

where  $\mathbf{X}$  is the input matrix, and  $\mathbf{W}_i^Q$ ,  $\mathbf{W}_i^K$ , and  $\mathbf{W}_i^V$  are learned weight matrices specific to the  $i$ -th head.

Each head calculates its attention output by first computing **scaled dot-product attention**, then passing the results through a **softmax function** to get **attention weights**. These weights are then applied to the values:

$$\text{Attention}_i(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) = \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^\top}{\sqrt{d_k}}\right) \mathbf{V}_i$$

where  $d_k$  is the dimensionality of the key vectors, used here for scaling.

### 7.17.2 Concatenation and Linear Transformation

After computing attention outputs from all heads, the results are concatenated into a single matrix and linearly transformed to produce the **final multi-head attention output**. If there are  $h$  heads, then the concatenated output is:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{Attention}_1, \text{Attention}_2, \dots, \text{Attention}_h) \mathbf{W}^O$$

where  $\mathbf{W}^O$  is a **learned weight matrix** for output transformation.

### 7.17.3 Advantages of Multi-Head Attention

The use of multiple heads in attention enhances the model's capacity to learn a diverse set of relationships within the input data. **Each head can focus on different parts of the sequence**, such as short-range dependencies or long-range dependencies, and capture distinct linguistic or structural patterns. This flexibility is essential in natural language processing tasks where understanding both immediate and distant context can improve model performance.

Moreover, multi-head attention contributes to greater model expressiveness, as each head's unique perspective helps capture complex interactions in the input data. In transformer models, this capability allows for more sophisticated language understanding, generating contextually rich representations of sequences.

### 7.17.4 Application in Transformer Models

In transformers, multi-head attention serves as a foundational component for both the encoder and decoder layers. It enables transformers to process input sequences in parallel rather than sequentially, which significantly speeds up training and inference. **Multi-head attention, combined with position encoding and feed-forward layers, has proven highly effective in achieving state-of-the-art performance** across various tasks, including language translation, sentiment analysis, and text summarization.

## 7.18 Softmax with Temperature

The softmax function is an **essential component** in neural network architectures for multi-class classification. By **transforming logits into probabilities**, softmax allows for interpretable and probabilistic predictions. Its differentiable nature, compatibility with cross-entropy loss, and ease of use in **backpropagation** make it an invaluable tool in the training and application of neural network models.

Softmax converts a vector of real numbers into a **probability distribution**, with each element representing the **likelihood** of a particular class. By introducing a **temperature parameter**  $T$ , the softmax function can be adjusted to make the probability distribution either **more “peaked” (decisive)** or **“smooth” (uncertain)**, depending on the needs of the application. This section will discuss the role of temperature in softmax, its mathematical formulation, and its practical applications.

### 7.18.1 Standard Softmax Function

The softmax function for a vector  $\mathbf{z} = [z_1, z_2, \dots, z_n]$  is defined as follows:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}$$

where  $e^{z_i}$  is the exponential of each element  $z_i$  in the input vector, and the denominator is the sum of exponentials of all elements in the vector, ensuring that the outputs sum to 1. This function transforms the input values into a probability distribution, highlighting the largest values more strongly.

### 7.18.2 Softmax with Temperature

Introducing a temperature parameter  $T$  modifies the softmax function to **allow control over the “sharpness” of the output distribution**. When  $T = 1$ , the function behaves as the standard softmax. However, adjusting  $T$  changes the effect of each input value’s exponential term. The temperature-scaled softmax is given by:

$$\text{softmax}(z_i, T) = \frac{e^{z_i/T}}{\sum_{j=1}^n e^{z_j/T}}$$

where  $T > 0$  is the temperature parameter.

Key behaviors of **temperature scaling** include:

- **Low Temperature ( $T < 1$ )**: The softmax output becomes more “peaked”, increasing the probability of the highest value in  $\mathbf{z}$  and decreasing the probability of other elements. This results in a **more decisive classification**, as it amplifies the differences between high and low values.

- **High Temperature ( $T > 1$ ):** The output becomes more "smooth", meaning probabilities are more evenly distributed across possible outcomes. This **reduces certainty in the classification**, which can be beneficial in exploratory settings or when avoiding overconfidence in uncertain conditions.

### 7.18.3 Applications of Softmax with Temperature

Temperature scaling has several applications, particularly in machine learning **tasks where model confidence needs to be adjusted or tuned**:

1. **Knowledge Distillation:** In knowledge distillation, a large "teacher" model transfers knowledge to a smaller "student" model. The teacher model's output is often softened using a higher temperature, giving the student model more informative gradients for training by spreading out the probability distribution.
2. **Reinforcement Learning:** In reinforcement learning, temperature scaling is used to control the exploration-exploitation trade-off. A higher temperature encourages exploration by smoothing out action probabilities, while a lower temperature promotes exploitation by focusing on the highest-probability actions.
3. **Natural Language Processing (NLP):** For tasks such as language generation, adjusting temperature can control the diversity of generated text. A high temperature produces more varied and creative outputs, while a low temperature yields more deterministic and likely coherent sentences.
4. **Uncertainty Quantification:** Temperature scaling is also utilized to calibrate model predictions, providing more realistic probabilities for each class. This is especially useful in applications where overconfidence in predictions can lead to errors, such as in healthcare or finance.

In summary, the temperature-scaled softmax function provides a flexible way to **control the behavior of probability distributions generated by neural networks**. By adjusting the temperature parameter, one can tune the model's certainty in predictions, aiding in tasks like knowledge distillation, exploration in reinforcement learning, and calibrated probability estimation. Temperature scaling is therefore a valuable tool for enhancing both model interpretability and performance in various applications.

## 7.19 Transformer Architecture

### 7.19.1 Introduction to Transformers

The Transformer architecture represents a significant advancement in natural language processing by **eliminating the need for recurrent layers** traditionally used in **sequential data processing**. Proposed by Vaswani et al. in 2017, the Transformer model introduced the concept of **self-attention** and shifted away from **recurrence-based mechanisms** to allow for parallel processing, thereby **overcoming limitations** in computational efficiency and long-term dependency handling.

#### 7.19.1.1 Motivation and Design Philosophy

The motivation for the Transformer model arose from the **need for a model that could process sequences in parallel while maintaining high performance in capturing long-range dependencies**. Unlike Recurrent Neural Networks (RNNs) or Long Short-Term Memory (LSTM) networks, which process data sequentially, Transformers apply attention mechanisms to each token in a **sequence simultaneously**. This parallelism reduces the computation time significantly, making it well-suited for large-scale data and modern computational hardware like **GPUs and TPUs**.

#### 7.19.1.2 Self-Attention Mechanism

The **core innovation** of the Transformer architecture is the **self-attention mechanism**, which allows each token in the input sequence to attend to every other token. Self-attention calculates a weighted representation of other tokens, helping each token capture relationships across the entire sequence. This mechanism involves **three key components: query, key, and value vectors**, which are used to compute **attention scores**. The weighted sum of these values forms the self-attention output, which is used to represent contextual dependencies.

#### 7.19.1.3 Parallel Processing and Scalability

By **eliminating recurrence**, the Transformer architecture **enables highly parallelizable computations**. This feature is essential for **processing long sequences**, where traditional RNN-based methods become computationally expensive. In the Transformer, **multiple self-attention layers and feed-forward layers** operate simultaneously, greatly reducing the training time on large datasets. Additionally, the architecture scales efficiently with the number of layers, making it highly suitable for large-scale language models.

#### 7.19.1.4 Impact and Applications

Since its introduction, the Transformer has become the **foundation of numerous NLP models**, including BERT, GPT, and T5. These models have achieved **state-of-the-art performance** across various NLP tasks, including machine translation, text summarization, and question answering. The ability of the Transformer to **capture complex linguistic features** through attention mechanisms has revolutionized NLP and has **enabled advancements** in other fields such as computer vision and protein modeling.

The Transformer architecture's combination of self-attention, scalability, and parallel processing has led to a paradigm shift in how sequence-based data is handled, demonstrating its versatility and power in numerous real-world applications.

### 7.19.2 Feed-Forward Networks

In the Transformer architecture, **each encoder and decoder block consists of a self-attention layer followed by a position-wise feed-forward network**. This feed-forward network provides additional depth and non-linearity to the model, enabling it to capture complex transformations after the attention mechanism has processed input dependencies. The feed-forward layers operate independently on each position within the sequence, applying the same linear transformations to each token.

#### 7.19.2.1 Architecture of Feed-Forward Networks

The feed-forward network in each encoder and decoder block is composed of **two linear layers with a ReLU (Rectified Linear Unit) activation function** between them. The **first linear layer expands the dimensionality** of the input, usually by a factor of four, creating a higher-dimensional representation that enhances the model's capacity to capture intricate patterns. The **second linear layer reduces the dimensionality back** to the original size, ensuring that the output from the feed-forward network matches the input dimension. This structure can be mathematically represented as:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (7.15)$$

where  $W_1$  and  $W_2$  are weight matrices, and  $b_1$  and  $b_2$  are bias terms. The non-linearity introduced by ReLU enhances the model's expressive power, allowing it to learn a variety of complex transformations.

#### 7.19.2.2 Role in Transformer Blocks

Within each encoder and decoder layer, the **feed-forward network follows the multi-head attention mechanism** and is applied independently to each token. This positioning allows the network to further **process and refine the context-rich representations**.

obtained from the self-attention layers. The **combination of self-attention and feed-forward layers in each block enables the model to capture both local and global dependencies**, making the Transformer highly effective for a variety of tasks, from translation to text summarization.

#### 7.19.2.3 Advantages of Position-Wise Feed-Forward Networks

Feed-forward networks add non-linear transformations, which aid in **capturing complex relationships beyond the linear interactions handled by the attention mechanism**. The **independence** of the feed-forward layer with respect to each position allows for efficient **parallelization**, a major advantage over sequential models like RNNs. Additionally, the separation of self-attention and feed-forward computations allows the Transformer model to handle large-scale data and long-range dependencies efficiently.

The feed-forward network, along with multi-head self-attention, forms the backbone of the Transformer architecture. Together, these components enable the model to perform high-level reasoning and manage sophisticated tasks in natural language processing.

#### 7.19.3 Introduction of Encoder and Decoder

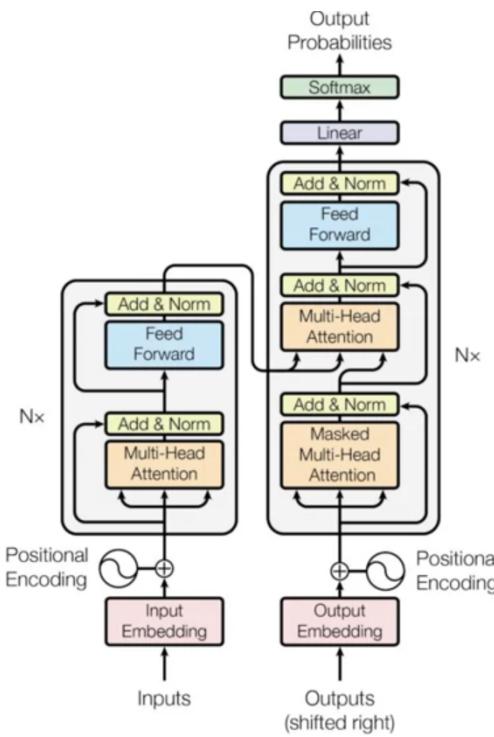


Figure 7.20: Transformer architecture

The encoder-decoder architecture represents a **pivotal advancement** in natural language processing (NLP), particularly in sequence-to-sequence tasks such as machine translation,

abstractive summarization, and question answering (Sutskever et al., 2014). This framework is built upon two primary components: an encoder and a decoder.

The Transformer model, which was introduced by Vaswani et al. (2017), is a cornerstone in sequence-to-sequence tasks. The Transformer architecture employs an encoder-decoder setup, each consisting of multiple identical layers with the specifics of its essential components.

### 7.19.3.1 Encoder

The encoder is responsible for processing the input sequence and compressing the information into a context or memory for the decoder.

Each encoder layer comprises **three main elements**:

- **Multi-Head Attention:** This component allows the model to **focus on different parts of the input** for each attention head, thereby capturing various aspects of the data.
- **Feed-Forward Neural Network:** A simple yet effective neural network that operates on the attention vectors, **applying nonlinear transformation and making it available** for the next encoder layer (and the decoder layer).
- **Add & Norm:** The Add & Norm layer aids in **stabilizing the activations** by combining residual connections and layer normalization, ensuring **smoother training and mitigating the vanishing gradient** problem in the encoder (and the decoder).

The input text is tokenized into units (words or sub-words), which are then embedded into **feature vectors**  $x_1, \dots, x_T$ . A unidirectional encoder updates its **hidden state**  $h_t$  at each time  $t$  using  $h_{t-1}$  and  $x_t$  as given by:

$$h_t = f(h_{t-1}, x_t) \quad (1)$$

The **final state**  $h_T$  of the encoder is known as the **context variable** or the **context vector**, and it encodes the information of the entire input sequence and is given by:

$$c = m(h_1, \dots, h_T) \quad (2)$$

where  $m$  is the **mapping function** and, in the simplest case, maps the context variable to the last hidden state

$$c = m(h_1, \dots, h_T) = h_T \quad (3)$$

Adding more complexity to the architecture, the encoders can be bidirectional; thus, the hidden state would not only depend on the previous hidden state  $h_{t-1}$  and input  $x_t$ , but also on the following state  $h_{t+1}$ .

## Example: Understanding the Encoder in Encoder-Decoder Architecture

To demonstrate how an encoder works in an encoder-decoder architecture, we will walk through a simple example using a sentence: “**I love NLP**”.

### Step 1: Tokenization

We begin by tokenizing the input sentence into individual words or subwords:

Sentence : “I love NLP”

Tokens : [“I”, “love”, “NLP”]

### Step 2: Embedding the Tokens

Each token is embedded into a fixed-dimensional vector. For simplicity, assume we use an embedding size of 4:

$$x_1 = [0.2, 0.5, 0.1, 0.3] \quad (\text{embedding of "I"})$$

$$x_2 = [0.4, 0.6, 0.3, 0.7] \quad (\text{embedding of "love"})$$

$$x_3 = [0.9, 0.2, 0.5, 0.4] \quad (\text{embedding of "NLP"})$$

### Step 3: Unidirectional Encoder (Equation 1)

We use a Recurrent Neural Network (RNN) encoder to process the input sequence. The encoder updates its hidden state at each time step using the formula:

$$h_t = f(h_{t-1}, x_t)$$

where:

- $h_t$ : hidden state at time  $t$
- $h_{t-1}$ : previous hidden state
- $x_t$ : current input vector
- $f$ : function representing the RNN cell (e.g., GRU, LSTM)

Assume the hidden state size is also 4 and initialize  $h_0 = [0, 0, 0, 0]$ .

1. For the first word  $I$ :

$$h_1 = f(h_0, x_1) = f([0, 0, 0, 0], [0.2, 0.5, 0.1, 0.3])$$

Assume the RNN cell produces:

$$h_1 = [0.1, 0.2, 0.1, 0.4]$$

2. For the second word *love*:

$$h_2 = f(h_1, x_2) = f([0.1, 0.2, 0.1, 0.4], [0.4, 0.6, 0.3, 0.7])$$

Assume the RNN produces:

$$h_2 = [0.3, 0.5, 0.4, 0.6]$$

3. For the third word *NLP*:

$$h_3 = f(h_2, x_3) = f([0.3, 0.5, 0.4, 0.6], [0.9, 0.2, 0.5, 0.4])$$

Assume the RNN produces:

$$h_3 = [0.6, 0.3, 0.5, 0.7]$$

**Step 4: Context Vector (Equation 2)** The final hidden state  $h_T = h_3$  represents the context vector  $c$ :

$$c = m(h_1, h_2, h_3) = h_3 = [0.6, 0.3, 0.5, 0.7]$$

In this unidirectional encoder, the context vector  $c$  is simply the last hidden state  $h_3$ .

### 7.19.3.2 Decoder

The decoder **takes the context from the encoder and generates the output sequence**. It is also composed of multiple layers and has many **commonalities** with the encoder, but with **minor changes**:

- **Masked Multi-Head Attention:** Similar to multi-head attention **but with a masking mechanism** to ensure that the **prediction for a given word doesn't depend on future words** in the sequence.
- **Encoder-Decoder Attention:** This layer allows the decoder to focus on relevant parts of the input sequence, leveraging the context provided by the encoder.
- **Feed-Forward Neural Network:** Identical in architecture to the one in the encoder, this layer further refines the attention vectors in preparation for generating the output sequence.

Upon obtaining the context vector  $c$  from the encoder, the decoder starts to generate the output sequence  $y = (y_1, y_2, \dots, y_U)$ , where  $U$  may differ from  $T$ . Similar to the encoder, the decoder's hidden state at any time  $t$  is given by:

$$s_{t'} = g(s_{t'-1}, y_{t'-1}, c) \quad (2.4)$$

The hidden state of the decoder flows to an output layer, and the conditional distribution of the next token at time  $t'$  is given by:

$$P(y_{t'}|y_{t'-1}, \dots, y_1, c) = \text{softmax}(s_{t'-1}, y_{t'-1}, c) \quad (2.5)$$

**Example:**

### Step 1: Decoder Initialization

Upon obtaining the context vector  $c$  from the encoder, the decoder starts generating the output sequence  $y = (y_1, y_2, \dots, y_U)$ . Note that  $U$ , the length of the output sequence, may differ from  $T$ , the length of the input sequence.

### Step 2: Decoder Hidden State (Equation 2.4)

The decoder updates its hidden state at each time step  $t'$  using the context vector and the previously generated token:

$$s_{t'} = g(s_{t'-1}, y_{t'-1}, c)$$

where:

- $s_{t'}$ : hidden state at time  $t'$
- $s_{t'-1}$ : previous hidden state
- $y_{t'-1}$ : token generated at the previous time step
- $c$ : context vector from the encoder

Assume the decoder's hidden state is initialized with  $s_0 = [0, 0, 0, 0]$ .

1. **Generating the first word:** The decoder takes the context vector  $c$  and a special start token **<start>** to generate the first hidden state:

$$s_1 = g(s_0, \text{<start>}, c)$$

Assume:

$$s_1 = [0.2, 0.4, 0.3, 0.5]$$

The hidden state  $s_1$  is passed through a softmax layer to generate the probability distribution over the target vocabulary:

$$P(y_1|\text{<start>}, c) = \text{softmax}(s_1, \text{<start>}, c)$$

Let's say the output token with the highest probability is "**J'aime**" (which means "I love" in French).

- 2. Generating the second word:** The decoder now takes the previous hidden state  $s_1$ , the previously generated token  $y_1 = \text{"J'aime"}$ , and the context vector  $c$  to generate the next hidden state:

$$s_2 = g(s_1, y_1, c) = g([0.2, 0.4, 0.3, 0.5], \text{"J'aime"}, [0.6, 0.3, 0.5, 0.7])$$

Assume:

$$s_2 = [0.5, 0.3, 0.6, 0.4]$$

The hidden state  $s_2$  is used to generate the next token:

$$P(y_2|y_1, c) = \text{softmax}(s_2, y_1, c)$$

Suppose the output token with the highest probability is **"NLP"**.

- 3. Generating the end token:** The process continues until the decoder generates an end token **<end>**. For the third step:

$$s_3 = g(s_2, y_2, c) = g([0.5, 0.3, 0.6, 0.4], \text{"NLP"}, [0.6, 0.3, 0.5, 0.7])$$

Assume:

$$s_3 = [0.7, 0.2, 0.4, 0.6]$$

The next word generated could be **"."** (a period) or the special end token **<end>**:

$$P(y_3|y_2, y_1, c) = \text{softmax}(s_3, y_2, c)$$

The **final generated sequence** is:

"J'aime NLP"

### Explanation of the Decoder Process

- The decoder uses the **context vector**  $c$  from the encoder as well as its previous hidden states and **previously generated tokens** to generate the **next token**.
- The **softmax function** is used to produce a probability distribution over the target vocabulary, allowing the decoder to select the **most likely** next word.
- This process continues until the decoder generates the end token **<end>**.

#### 7.19.3.3 Training and Optimization

The encoder-decoder model is trained end-to-end through supervised learning. The **standard loss function** employed is the **categorical cross-entropy** between the predicted output sequence and the actual output. This can be represented as:

$$\mathcal{L} = - \sum_{t=1}^U \log p(y_t|y_{t-1}, \dots, y_1, c) \quad (2.6)$$

Optimization of the model parameters typically employs **gradient descent variants, such as the Adam or RMSprop algorithms**.

#### 7.19.3.4 Issues with Encoder-Decoder Architectures

As outlined in the preceding section, the encoder component condenses the information from the source sentence into a singular context variable  $c$  for subsequent utilization by the decoder. Such a reductionist approach inherently suffers from information loss, particularly as the input length increases.

Moreover, natural language's syntactic and semantic intricacies often entail long-range dependencies between tokens, which are challenging to encapsulate effectively within a singular context vector.

However, it should be noted that the hidden states at each time step in the encoder contain valuable information that remains available for the decoder's operations. These hidden states can exert variable influence on each decoding time step, thereby partially alleviating the limitations of a singular context variable. Nevertheless, Recurrent Neural Networks (RNNs), the foundational architecture for many encoder-decoder models, have shortcomings, such as susceptibility to vanishing and exploding gradients (Hochreiter, 1998). Additionally, the sequential dependency intrinsic to RNNs complicates parallelization, thereby imposing computational constraints.

#### 7.19.3.5 Tokenization and Representation

In Transformer models, **tokenization** typically **converts sentences into a machine-readable format**. This can be done at the level of words or subwords, depending on the **granularity** required for the specific application. Each word in the sentence is treated as a distinct token in word-level tokenization. These **tokens are then mapped to their corresponding vector representations**, such as word embeddings, which serve as the input to the Transformer model.

This approach may face **limitations when dealing with out-of-vocabulary words**. **Subword-level approaches** such as byte-pair encoding (**BPE**) or **WordPiece** often address the limitations of word-level tokenization. In these methods, words are broken down into **smaller pieces or subwords**, providing a way to represent out-of-vocabulary terms and capture **morphological nuances**. These subwords are then mapped to embeddings and fed into the Transformer. For instance, the word “unhappiness” could be split into subwords such as “un” and “happiness”. These subwords are then individually mapped to their embeddings. This method increases the model's ability to **generalize and handle a broader range of vocabulary**, including words not seen during training.

A hybrid approach combining **word and subword-level tokenization** can also leverage both. Such a strategy **balances** the comprehensiveness of subword-level representations with the interpretability of word-level tokens.

#### 7.19.3.6 Positional Encoding

Since the Transformer model processes **all tokens in the input sequence in parallel**, it does not have a built-in mechanism to account for the token positions or order. Positional encoding is introduced to provide the model with information about the **relative positions of the tokens** in the sequence. The positional encoding is usually added to the input embeddings before they are fed into the Transformer model.

If the length of the sentence is given by  $l$  and the embedding dimension/depth is given by  $d$ , positional encoding  $\mathbf{P}$  is a 2-dimensional matrix of the same dimension, i.e.,  $\mathbf{P} \in \mathbb{R}^{l \times d}$ . Every position can be represented with the equation in terms of  $i$ , which is along the  $l$ , and  $j$ , which is along the  $d$  dimension as

$$\mathbf{P}_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right)$$

$$\mathbf{P}_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right)$$

for  $i = 0, \dots, l - 1$  and  $j = 0, \dots, \lfloor (d - 1)/2 \rfloor$ .

The function definition above indicates that the **frequencies decrease along the vector dimension** and form a **geometric progression** from  $2\pi$  to  $10000 \cdot 2\pi$  on the wavelengths. For  $d = 512$  dimensions for a maximum positional length of  $l = 100$ , the positional encoding visualization is shown.

## 7.20 Autoencoder

### Notation

Symbol	Meaning
$\mathcal{D}$	The dataset, $\mathcal{D} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n)}\}$ , containing $n$ data samples; $ \mathcal{D}  = n$ .
$\mathbf{x}^{(i)}$	Each data point is a vector of $d$ dimensions, $\mathbf{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}, \dots, x_d^{(i)}]$ .
$\mathbf{x}$	A single data sample from the dataset, $\mathbf{x} \in \mathcal{D}$ .
$\mathbf{x}'$	The reconstructed version of $\mathbf{x}$ .
$\tilde{\mathbf{x}}$	The corrupted version of $\mathbf{x}$ .
$\mathbf{z}$	The compressed code learned in the bottleneck layer.
$a_j^{(l)}$	The activation function for the $j$ -th neuron in the $l$ -th hidden layer.
$g_\phi(\cdot)$	The <b>encoding</b> function parameterized by $\phi$ .
$f_\theta(\cdot)$	The <b>decoding</b> function parameterized by $\theta$ .
$q_\phi(\mathbf{z} \mathbf{x})$	Estimated posterior probability function, also known as <b>probabilistic encoder</b> .
$p_\theta(\mathbf{x} \mathbf{z})$	Likelihood of generating a true data sample given the latent code, also known as <b>probabilistic decoder</b> .

Table 7.3: Notations used in Autoencoders

### Definition

Autoencoders are a type of **Unsupervised Neural Network**. Their operation is described as follows:

- Receive an input dataset.
- Transform the Input Data into a different representation in the **Latent Space**.
- Reconstruct the Input Data from the **Latent Space Representation**.

In terms of structure, an Autoencoder model consists of two components (**subnetworks**):

- **Encoder:** Receives Input Data and transforms it into a different form in the Latent Space.

$$s = E(x)$$

Here,  $x$  is the Input Data,  $E$  is the Encoder, and  $s$  is the Output in the Latent Space.

- **Decoder:** Receives the Encoder's Output in the Latent Space,  $s$ , and reconstructs the Input Data.

$$o = D(s)$$

Here,  $o$  is the Output of the Decoder  $D$ .

The **overall formula** for the entire Autoencoder process is:

$$o = D(E(x))$$

The **general architecture** of Autoencoders is illustrated as follows:

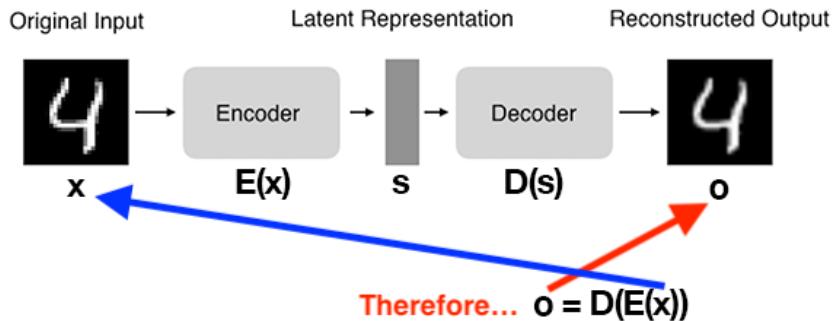


Figure 7.21: Autoencoders architecture

To train an Autoencoder model, we provide it with Input Data. It attempts to **reconstruct the Input Data by minimizing the Mean Squared Error** between the model's Output and the Input Data. In other words, the Input Data serves as its own label. An ideal Autoencoder model is one where its Output is identical to its Input Data.

You might wonder **why we need to create an Autoencoder model**, going through a process just to reconstruct the Input Data. **Why not simply copy the Input Data directly?**

⇒ Because the **real value** of an Autoencoder model lies in the **Encoder's Output in the Latent Space**. In other words, we only care about the Encoder and the Latent Space, not so much about the Decoder.

### A simple example to illustrate:

Suppose we have a **set of images**, each with dimensions **28x28x1**, meaning we need  $28 \times 28 \times 1 = 784$  bytes to store each image.

Using an Autoencoder model, we transform that image into a smaller vector of only 16 bytes in the Latent Space.

Using this 16-byte vector, we can **reconstruct the original image with up to 98% similarity**. This helps us **save a lot of storage space**, especially when transmitting data over the Internet. This is one of the applications of Autoencoders.

### 7.20.1 Applications of Autoencoders

The general architecture of Autoencoders is illustrated as follows:

Some applications of Autoencoders in the field of Computer Vision include:

- **Dimensionality Reduction:** This application is similar to the **PCA** algorithm but is more effective.
- **Denoising:** Reduces noise in data, a step in **image preprocessing**, which helps improve the **accuracy of OCR systems**.
- **Anomaly/outlier Detection:** Identifies **abnormal data points** in a dataset, such as mislabeled data or data that deviates from the overall distribution. Once anomalies are detected, they can be **removed or the model retrained to improve accuracy**.

In the field of Natural Language Processing (**NLP**), Autoencoder models help solve tasks such as:

- Generating text descriptions for images (**Image Caption Generation**).
- Summarizing text content (**Text Summarization**).
- Extracting features (**Word Embedding**).

### 7.20.2 Practical Example: Using Autoencoders for Text Denoising

Let's walk through a simple use case where Autoencoders are used to **denoise text data**.

Assume you have a dataset **containing sentences that are noisy due to typos and formatting errors**. The **goal** is to train a Denoising Autoencoder (DAE) to **remove** this noise and **reconstruct** clean sentences.

**Autoencoder** is a neural network designed to learn an identity function in an unsupervised manner to reconstruct the original input while compressing the data in the process, discovering a more efficient and compressed representation. The idea originated in the **1980s** and was later promoted by the seminal paper by **Hinton & Salakhutdinov, 2006**.

It consists of **two networks**:

- **Encoder network:** Translates the original high-dimensional input into the latent low-dimensional code. The input size is larger than the output size.
- **Decoder network:** Recovers the data from the code, typically with larger and larger output layers.

The encoder network essentially accomplishes **dimensionality reduction**, similar to Principal Component Analysis (**PCA**) or Matrix Factorization (**MF**). Additionally, the autoencoder is explicitly optimized for data reconstruction from the code. A good intermediate representation can capture latent variables and benefit a full **decompression** process.

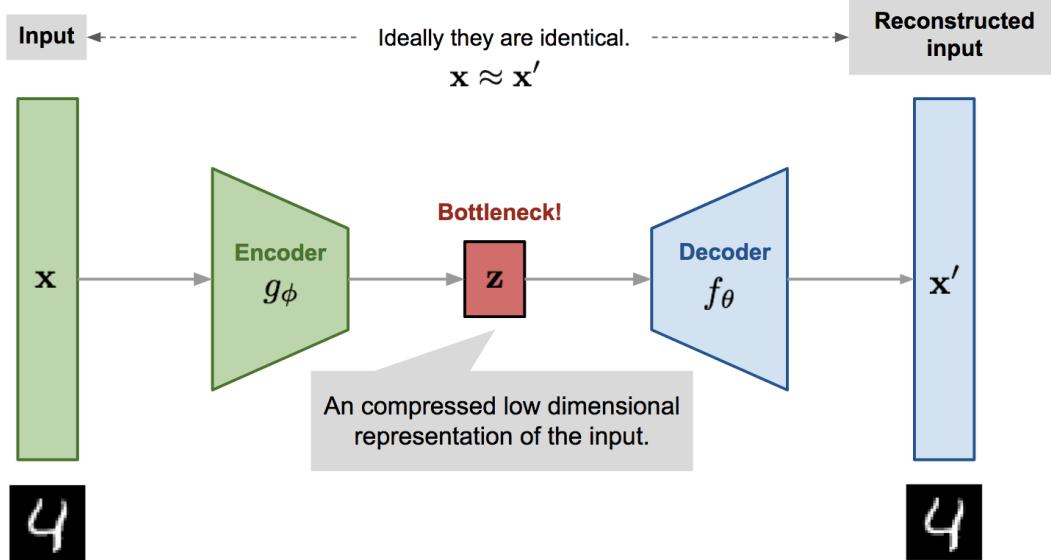


Figure 7.22: Illustration of Autoencoder

The model contains an **encoder function**  $g(\cdot)$  parameterized by  $\phi$  and a decoder function  $f(\cdot)$  parameterized by  $\theta$ . The **low-dimensional code** learned for input  $\mathbf{x}$  in the **bottleneck layer** is  $\mathbf{z} = g_\phi(\mathbf{x})$ , and the reconstructed input is  $\mathbf{x}' = f_\theta(g_\phi(\mathbf{x}))$ .

The parameters  $(\theta, \phi)$  are learned together to output a reconstructed data sample close to the original input,  $\mathbf{x} \approx f_\theta(g_\phi(\mathbf{x}))$ . **Various metrics** can quantify the difference between two vectors, such as cross-entropy when using a sigmoid activation function, or simply **mean squared error (MSE) loss**:

$$L_{AE}(\theta, \phi) = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}^{(i)} - f_\theta(g_\phi(\mathbf{x}^{(i)})))^2$$

### 7.20.3 Denoising Autoencoder

Since the autoencoder learns the **identity function**, there is a risk of *overfitting* when there are **more network parameters than data points**.

To avoid **overfitting** and improve **robustness**, **Denoising Autoencoder** (Vincent et al., 2008) introduces noise to the input vector  $\tilde{\mathbf{x}} \sim \mathcal{M}_{\mathcal{D}}(\tilde{\mathbf{x}}|\mathbf{x})$ , where the model is trained to **recover the original input**:

$$\begin{aligned} \tilde{\mathbf{x}}^{(i)} &\sim \mathcal{M}_{\mathcal{D}}(\tilde{\mathbf{x}}^{(i)}|\mathbf{x}^{(i)}) \\ L_{DAE}(\theta, \phi) &= \frac{1}{n} \sum_{i=1}^n (\mathbf{x}^{(i)} - f_\theta(g_\phi(\tilde{\mathbf{x}}^{(i)})))^2 \end{aligned}$$

This design is **inspired by the fact that humans can recognize objects or scenes even**

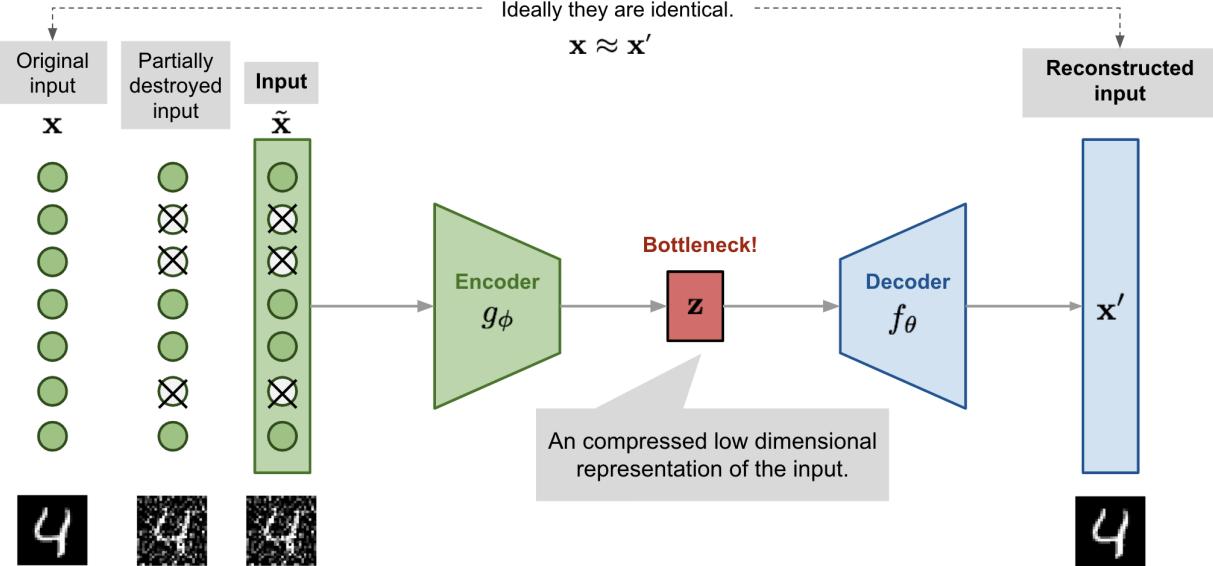


Figure 7.23: Illustration of denoising autoencoder model architecture.

**if partially occluded.** The denoising autoencoder learns to repair the partially corrupted input, capturing relationships between dimensions to infer missing information. This process helps in learning **robust** latent representations.

#### 7.20.4 Sparse Autoencoder

**Sparse Autoencoder** applies a **sparse** constraint on the hidden unit activations to avoid overfitting and improve robustness. It enforces that only a small number of hidden units are activated at the same time.

Let's define  $s_l$  as the number of neurons in the  $l$ -th hidden layer. The activation function for the  $j$ -th neuron is labeled  $a_j^{(l)}(\cdot)$ , where  $j = 1, \dots, s_l$ . The fraction of activation  $\hat{\rho}_j$  is expected to be a small value  $\rho$ , known as the **sparsity parameter**, typically  $\rho = 0.05$ :

$$\hat{\rho}_j^{(l)} = \frac{1}{n} \sum_{i=1}^n [a_j^{(l)}(\mathbf{x}^{(i)})] \approx \rho$$

This constraint is achieved by **adding a penalty term to the loss function**. The **KL-divergence**  $D_{\text{KL}}$  measures the **difference between two Bernoulli distributions** with means  $\rho$  and  $\hat{\rho}_j^{(l)}$ . The **hyperparameter**  $\beta$  controls the **strength of the sparsity penalty**:

$$\begin{aligned} \mathcal{L}_{\text{SAE}}(\theta) &= \mathcal{L}(\theta) + \beta \sum_{\ell=1}^L \sum_{j=1}^{s_\ell} D_{\text{KL}}(\rho \parallel \hat{\rho}_j^{(\ell)}) \\ &= \mathcal{L}(\theta) + \beta \sum_{\ell=1}^L \sum_{j=1}^{s_\ell} \rho \log \frac{\rho}{\hat{\rho}_j^{(\ell)}} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}_j^{(\ell)}} \end{aligned}$$

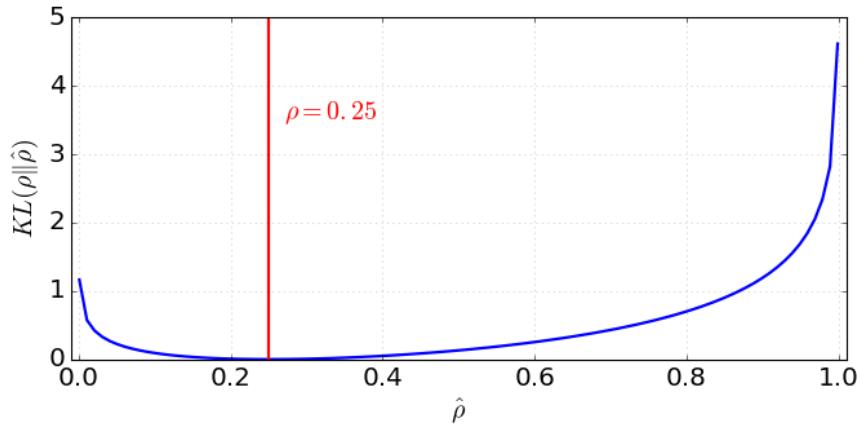


Figure 7.24: KL divergence between a Bernoulli distribution with mean  $\rho = 0.25$  and a Bernoulli distribution with mean  $0 \leq \hat{\rho} \leq 1$ .

### 7.20.5 $k$ -Sparse Autoencoder

In  **$k$ -Sparse Autoencoder**, the sparsity is enforced by keeping only the **top- $k$  highest activations** in the bottleneck layer:

- Compute the compressed code:  $\mathbf{z} = g(\mathbf{x})$
- Sort the values in  $\mathbf{z}$  and keep only the top- $k$  values while setting others to zero:  $\mathbf{z}' = \text{Sparsify}(\mathbf{z})$
- Compute the output and loss:  $L = \|\mathbf{x} - f(\mathbf{z}')\|_2^2$

The **back-propagation** only updates the top- $k$  activated hidden units.

#### Data Preprocessing

First, we preprocess the text data by tokenizing sentences and converting them into numerical representations (e.g., using word embeddings). We then introduce noise by randomly swapping or deleting characters in the sentences.

Listing 7.3: Function add noise

```
# Example: Adding noise to text data
import random

def add_noise(sentence):
    noisy_sentence = list(sentence)
    num_noisy_chars = int(0.1 * len(sentence))
    for _ in range(num_noisy_chars):
        index = random.randint(0, len(noisy_sentence) - 1)
        noisy_sentence[index] = random.choice('abcdefghijklmnopqrstuvwxyz')
    return ''.join(noisy_sentence)
```

## Training the Denoising Autoencoder

Next, we train a simple Autoencoder model using TensorFlow. The input is the noisy text, and the target output is the clean text.

Listing 7.4: Training the Denoising Autoencoder

```
from tensorflow.keras.layers import Input, LSTM, Dense
from tensorflow.keras.models import Model

input_text = Input(shape=(None, 300)) # Assume 300-dimensional word
embeddings
encoded = LSTM(128)(input_text)
decoded = LSTM(300, return_sequences=True)(encoded)

autoencoder = Model(input_text, decoded)
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.fit(noisy_text, clean_text, epochs=20, batch_size=64)
```

## Evaluating the Model

After training, we evaluate the model by feeding it noisy sentences and comparing the reconstructed output to the original clean sentences. The goal is to minimize the Mean Squared Error (MSE) between the predicted and original sentences.

### 7.20.6 Variational Autoencoder (VAE)

The idea of **Variational Autoencoder** (Kingma & Welling, 2014), short for **VAE**, is actually less similar to all the autoencoder models above, but deeply rooted in the methods of variational Bayesian inference and graphical models.

Instead of mapping the input into a *fixed* vector, we want to map it into a distribution. Let's label this distribution as  $p_\theta$ , parameterized by  $\theta$ . The relationship between the data input  $\mathbf{x}$  and the latent encoding vector  $\mathbf{z}$  can be fully defined by:

- Prior  $p_\theta(\mathbf{z})$
- Likelihood  $p_\theta(\mathbf{x}|\mathbf{z})$
- Posterior  $p_\theta(\mathbf{z}|\mathbf{x})$

Assuming that we know the real parameter  $\theta^*$  for this distribution, in order to generate a sample that looks like a real data point  $\mathbf{x}^{(i)}$ , we follow these steps:

1. First, sample a  $\mathbf{z}^{(i)}$  from a prior distribution  $p_{\theta^*}(\mathbf{z})$ .
2. Then generate a value  $\mathbf{x}^{(i)}$  from a conditional distribution  $p_{\theta^*}(\mathbf{x}|\mathbf{z} = \mathbf{z}^{(i)})$ .

The optimal parameter  $\theta^*$  is the one that maximizes the probability of generating real data samples:

$$\theta^* = \arg \max_{\theta} \prod_{i=1}^n p_{\theta}(\mathbf{x}^{(i)})$$

Commonly, we use the log probabilities to convert the product on the right-hand side to a sum:

$$\theta^* = \arg \max_{\theta} \sum_{i=1}^n \log p_{\theta}(\mathbf{x}^{(i)})$$

Now, let's update the equation to better demonstrate the data generation process by involving the encoding vector:

$$p_{\theta}(\mathbf{x}^{(i)}) = \int p_{\theta}(\mathbf{x}^{(i)} | \mathbf{z}) p_{\theta}(\mathbf{z}) d\mathbf{z}$$

Unfortunately, it is not easy to compute  $p_{\theta}(\mathbf{x}^{(i)})$  in this way, as it is very expensive to check all the possible values of  $\mathbf{z}$  and sum them up. To narrow down the value space to facilitate faster search, we introduce a new approximation function,  $q_{\phi}(\mathbf{z}|\mathbf{x})$ , parameterized by  $\phi$ , to output a likely code given an input  $\mathbf{x}$ .

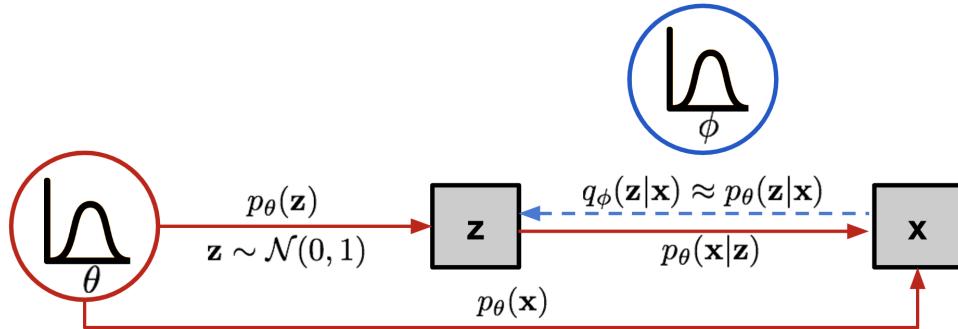


Figure 7.25: The graphical model involved in Variational Autoencoder. Solid lines denote the generative distribution  $p_{\theta}(\cdot)$  and dashed lines denote the distribution  $q_{\phi}(\mathbf{z}|\mathbf{x})$  to approximate the intractable posterior  $p_{\theta}(\mathbf{z}|\mathbf{x})$ .

Now, the structure looks a lot like an autoencoder:

- The conditional probability  $p_{\theta}(\mathbf{x}|\mathbf{z})$  defines a generative model, similar to the decoder  $f_{\theta}(\mathbf{x}|\mathbf{z})$  introduced earlier.  $p_{\theta}(\mathbf{x}|\mathbf{z})$  is also known as a *probabilistic decoder*.
- The approximation function  $q_{\phi}(\mathbf{z}|\mathbf{x})$  is the *probabilistic encoder*, playing a similar role as  $g_{\phi}(\mathbf{z}|\mathbf{x})$ .

#### 7.20.6.1 Loss Function: ELBO

The estimated posterior  $q_{\phi}(\mathbf{z}|\mathbf{x})$  should be very close to the real one  $p_{\theta}(\mathbf{z}|\mathbf{x})$ . We can use [Kullback-Leibler divergence](#) to quantify the distance between these two distributions. KL

divergence  $D_{\text{KL}}(X|Y)$  measures how much information is lost if the distribution  $Y$  is used to represent  $X$ .

In our case, we want to minimize  $D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}|\mathbf{x}))$  with respect to  $\phi$ .

But why use  $D_{\text{KL}}(q_\phi||p_\theta)$  (reversed KL) instead of  $D_{\text{KL}}(p_\theta||q_\phi)$  (forward KL)? Eric Jang has a great explanation in his [post](#) on Bayesian Variational methods. As a quick recap:

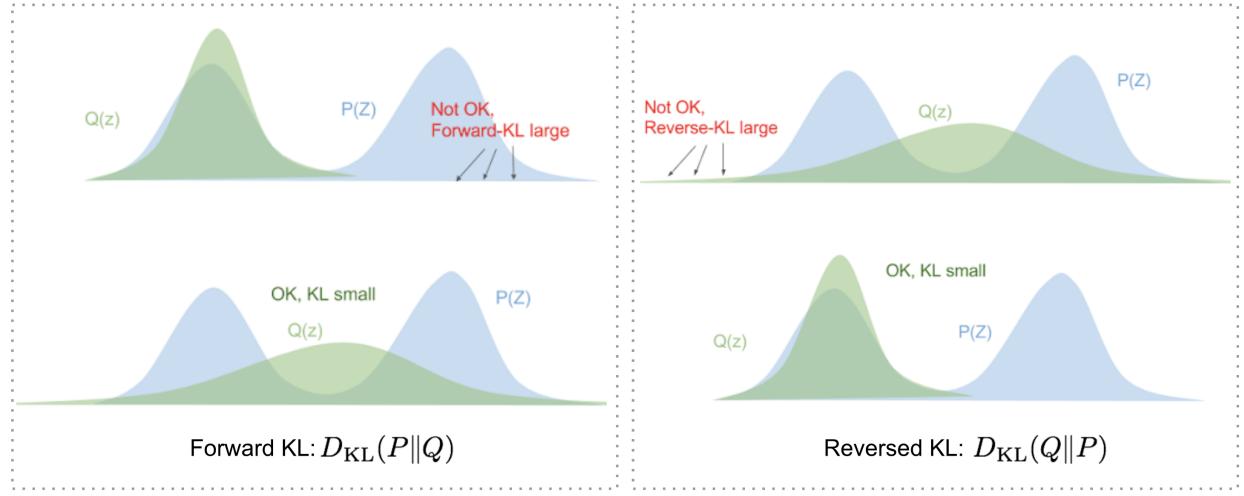


Figure 7.26: Forward and reversed KL divergence have different demands on how to match two distributions <sup>1</sup>.

- **Forward KL divergence:**  $D_{\text{KL}}(P||Q) = \mathbb{E}_{z \sim P(z)} \log \frac{P(z)}{Q(z)}$ ; we have to ensure that  $Q(z) > 0$  wherever  $P(z) > 0$ . The optimized variational distribution  $q(z)$  has to cover the entire  $p(z)$ .
- **Reversed KL divergence:**  $D_{\text{KL}}(Q||P) = \mathbb{E}_{z \sim Q(z)} \log \frac{Q(z)}{P(z)}$ ; minimizing the reversed KL divergence squeezes the  $Q(z)$  under  $P(z)$ .

Let's now expand the equation:

$$\begin{aligned}
& D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}|\mathbf{x})) \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}|\mathbf{x})} d\mathbf{z} \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{q_\phi(\mathbf{z}|\mathbf{x}) p_\theta(\mathbf{x})}{p_\theta(\mathbf{z}, \mathbf{x})} d\mathbf{z} \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \left( \log p_\theta(\mathbf{x}) + \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}, \mathbf{x})} \right) d\mathbf{z} \\
&= \log p_\theta(\mathbf{x}) + \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}, \mathbf{x})} d\mathbf{z} \\
&= \log p_\theta(\mathbf{x}) + \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{x}|\mathbf{z}) p_\theta(\mathbf{z})} d\mathbf{z} \\
&= \log p_\theta(\mathbf{x}) + \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \left[ \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z})} - \log p_\theta(\mathbf{x}|\mathbf{z}) \right] \\
&= \log p_\theta(\mathbf{x}) + D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})) - \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z})
\end{aligned}$$

So we have:

$$D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}|\mathbf{x})) = \log p_\theta(\mathbf{x}) + D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})) - \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z})$$

Once we rearrange the left and right-hand sides of the equation:

$$\log p_\theta(\mathbf{x}) - D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}|\mathbf{x})) = \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) - D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}))$$

The LHS of the equation is exactly what we want to maximize when learning the true distributions: we want to maximize the (log-)likelihood of generating real data (i.e.,  $\log p_\theta(\mathbf{x})$ ) while minimizing the difference between the real and estimated posterior distributions (the term  $D_{\text{KL}}$  acts like a regularizer). Note that  $p_\theta(\mathbf{x})$  is fixed with respect to  $q_\phi$ .

The negation of the above defines our loss function:

$$\begin{aligned}
L_{\text{VAE}}(\theta, \phi) &= -\log p_\theta(\mathbf{x}) + D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}|\mathbf{x})) \\
&= -\mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) + D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})) \\
\theta^*, \phi^* &= \arg \min_{\theta, \phi} L_{\text{VAE}}
\end{aligned}$$

In Variational Bayesian methods, this loss function is known as the *variational lower bound*, or *evidence lower bound* (ELBO). The "lower bound" part in the name comes from the fact that KL divergence is always non-negative, thus  $-L_{\text{VAE}}$  is the lower bound of  $\log p_\theta(\mathbf{x})$ :

$$-L_{\text{VAE}} = \log p_\theta(\mathbf{x}) - D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}|\mathbf{x})) \leq \log p_\theta(\mathbf{x})$$

Therefore, by minimizing the loss, we are maximizing the lower bound of the probability of generating real data samples.

### 7.20.6.2 Reparameterization Trick

The expectation term in the loss function involves generating samples from  $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$ . Sampling is a stochastic process, and therefore, we cannot backpropagate the gradient. To make it trainable, the *reparameterization trick* is introduced: It is often possible to express the random variable  $\mathbf{z}$  as a deterministic variable  $\mathbf{z} = \mathcal{T}_\phi(\mathbf{x}, \epsilon)$ , where  $\epsilon$  is an auxiliary independent random variable, and the transformation function  $\mathcal{T}_\phi$  parameterized by  $\phi$  converts  $\epsilon$  to  $\mathbf{z}$ .

For example, a common choice for  $q_\phi(\mathbf{z}|\mathbf{x})$  is a multivariate Gaussian with a diagonal covariance structure:

$$\begin{aligned}\mathbf{z} &\sim q_\phi(\mathbf{z}|\mathbf{x}^{(i)}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{2(i)} \mathbf{I}) \\ \mathbf{z} &= \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \epsilon, \quad \text{where } \epsilon \sim \mathcal{N}(0, \mathbf{I})\end{aligned}$$

where  $\odot$  refers to element-wise multiplication.

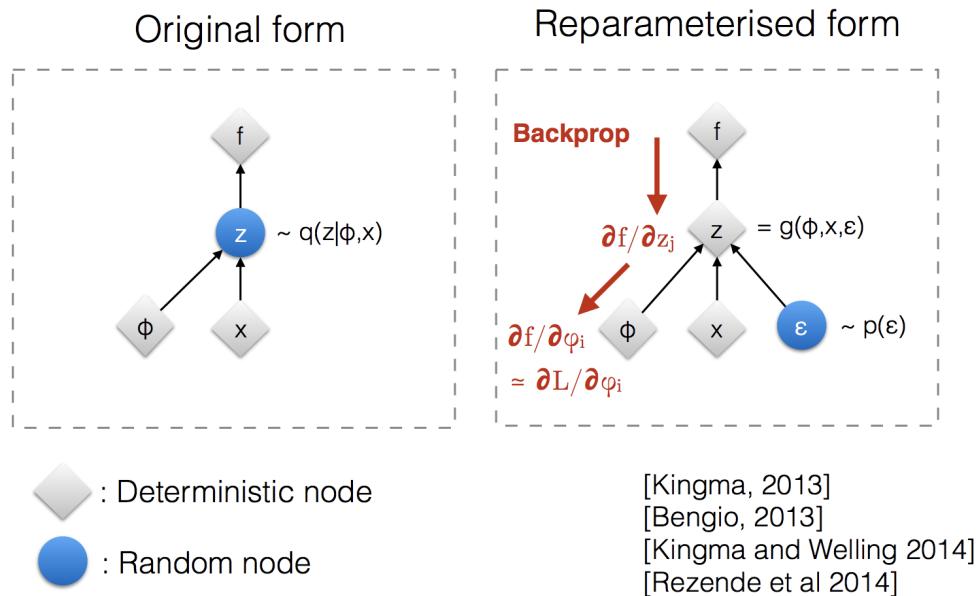


Figure 7.27: Illustration of how the reparameterization trick makes the  $\mathbf{z}$  sampling process trainable <sup>2</sup>.

The reparameterization trick works for other types of distributions too, not just Gaussian. In the multivariate Gaussian case, we make the model trainable by learning the mean and variance of the distribution,  $\mu$  and  $\sigma$ , explicitly using the reparameterization trick, while the stochasticity remains in the random variable  $\epsilon \sim \mathcal{N}(0, \mathbf{I})$ .

### 7.20.6.3 Beta-VAE

If each variable in the inferred latent representation  $\mathbf{z}$  is only sensitive to a single generative factor and relatively invariant to other factors, we say that this representation is *disentangled*.

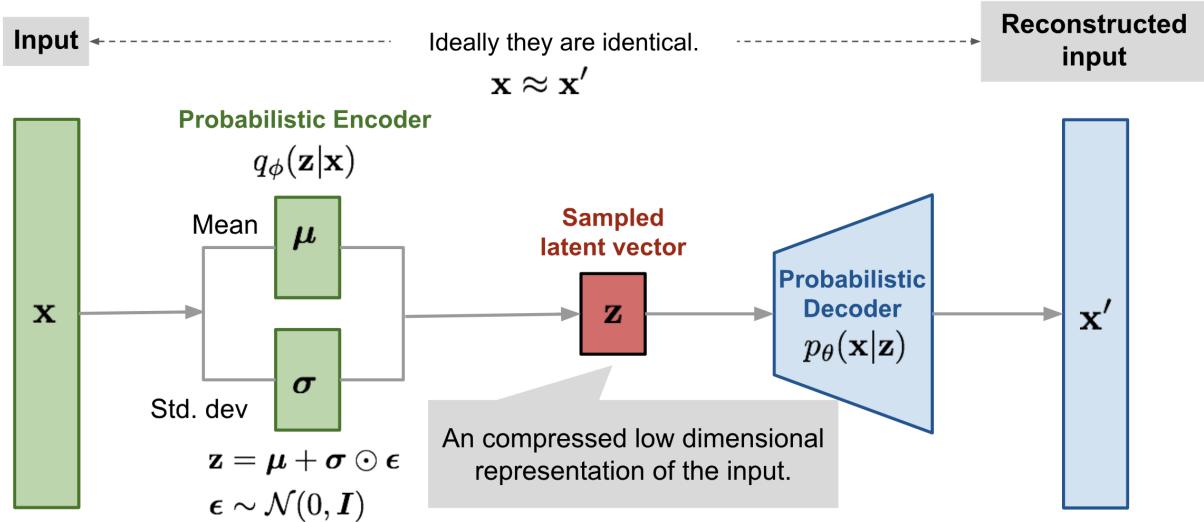


Figure 7.28: Illustration of the variational autoencoder model with the multivariate Gaussian assumption.

gled or *factorized*. One benefit that often comes with disentangled representation is *good interpretability* and easy generalization to a variety of tasks.

For example, a model trained on photos of human faces might capture attributes like gender, skin color, hair color, hair length, emotion, and whether the person is wearing glasses, among other relatively independent factors, in separate dimensions. Such a disentangled representation is very beneficial for facial image generation.

$\beta$ -VAE (Higgins et al., 2017) is a modification of the Variational Autoencoder with a special emphasis on discovering disentangled latent factors. Following the same incentive as VAE, we want to maximize the probability of generating real data, while keeping the distance between the real and estimated posterior distributions small (say, under a small constant  $\delta$ ):

$$\begin{aligned} & \max_{\phi, \theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} [\mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z})] \\ & \text{subject to } D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})) < \delta \end{aligned}$$

We can rewrite it as a Lagrangian with a Lagrange multiplier  $\beta$  under the **KKT condition**. The above optimization problem with only one inequality constraint is equivalent to maximizing the following equation  $\mathcal{F}(\theta, \phi, \beta)$ :

$$\begin{aligned} \mathcal{F}(\theta, \phi, \beta) &= \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) - \beta (D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})) - \delta) \\ &= \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) - \beta D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})) + \beta \delta \\ &\geq \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) - \beta D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})) \quad (\text{since } \beta, \delta \geq 0) \end{aligned}$$

The loss function of  $\beta$ -VAE is defined as:

$$L_{\text{BETA}}(\phi, \theta, \beta) = -\mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) + \beta D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}))$$

where the Lagrange multiplier  $\beta$  is considered a hyperparameter.

Since the negation of  $L_{\text{BETA}}(\phi, \theta, \beta)$  is the lower bound of the Lagrangian  $\mathcal{F}(\theta, \phi, \beta)$ , minimizing the loss is equivalent to maximizing the Lagrangian and thus solves our initial optimization problem.

When  $\beta = 1$ , it is equivalent to the original VAE. When  $\beta > 1$ , it imposes a stronger constraint on the latent bottleneck, limiting the representation capacity of  $\mathbf{z}$ . For some conditionally independent generative factors, keeping them disentangled is the most efficient representation. Therefore, a higher  $\beta$  encourages more efficient latent encoding and further promotes disentanglement. However, a higher  $\beta$  may introduce a trade-off between reconstruction quality and the degree of disentanglement.

Burgess, et al. (2017) discussed disentanglement in  $\beta$ -VAE in depth, inspired by the **information bottleneck theory**, and further proposed a modification to  $\beta$ -VAE to better control the encoding representation capacity.

## 7.21 Practical NLP Tasks and Applications

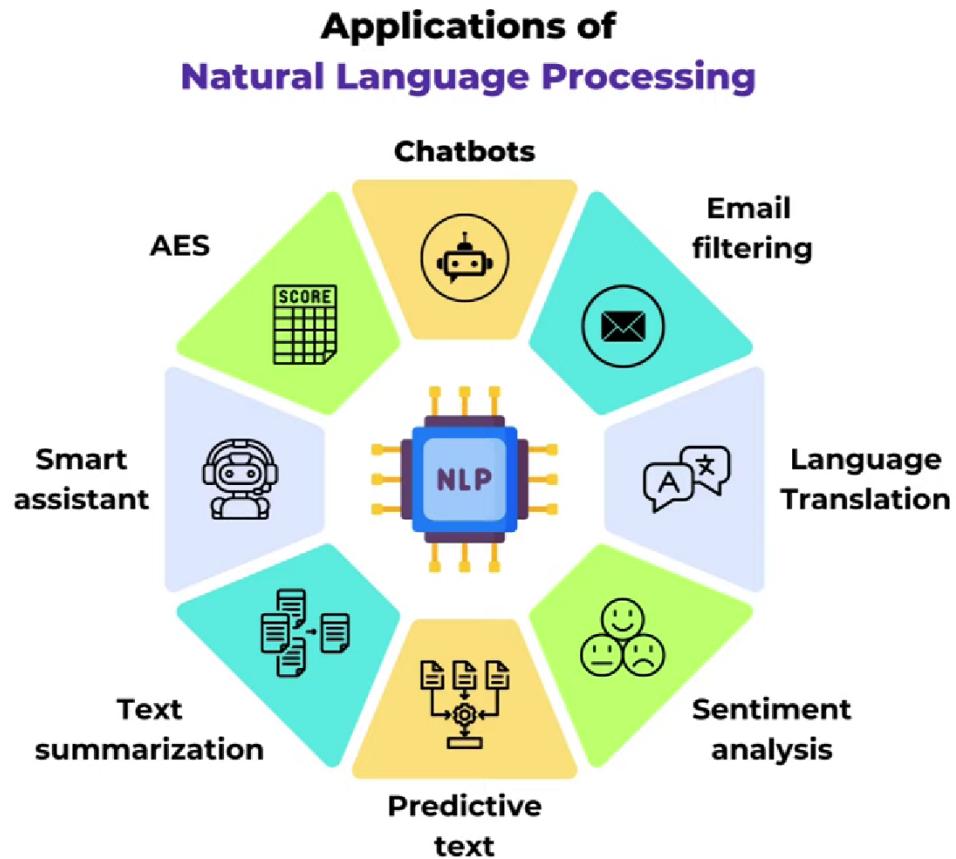


Figure 7.29: Application of Natural Language Processing

### 7.21.1 Sentence-Level Tasks

Natural language processing (NLP) at the **sentence level** encompasses several fundamental tasks that focus on understanding the **meaning, structure, and semantics** of individual sentences. These sentence-level tasks are essential in many practical applications, providing insights that contribute to larger language processing systems.

- **Natural Language Inference (NLI):**

Natural language inference, also known as **recognizing textual entailment (RTE)**, is the task of determining if a “hypothesis” sentence logically follows from a “premise” sentence. NLI classifies the relationship between the premise and hypothesis into one of **three categories**: **entailment** (the hypothesis is true given the premise), **contradiction** (the hypothesis is false given the premise), or **neutral** (the relationship cannot be determined). This task is **crucial** in applications such as question answering,

information retrieval, and dialogue systems, where understanding relationships between statements is essential.

- **Paraphrasing:**

Paraphrasing is the task of **rephrasing a sentence while retaining its original meaning**. This involves generating a sentence that conveys the same information as the input sentence but utilizes different words or structure. Paraphrasing is particularly valuable for text generation, content summarization, and question answering, as it allows NLP models to produce diverse expressions of the same content. **Encoder-decoder architectures**, often employing **transformer-based models**, are widely used to generate high-quality paraphrases.

- **Sentiment Analysis:**

Sentiment analysis involves identifying the **emotional tone** of a sentence, typically categorizing it as positive, negative, or neutral. This task has numerous applications, including social media monitoring, product review analysis, and customer feedback systems. Sentiment analysis can also be fine-grained, detecting specific emotions such as happiness, anger, or sadness. Models used for sentiment analysis range from **traditional machine learning approaches**, such as support vector machines and logistic regression, to **deep learning techniques**, including convolutional neural networks (CNNs) and recurrent neural networks (RNNs), which can capture complex language cues indicative of sentiment.

Each of these sentence-level tasks enables a more comprehensive understanding of language, supporting practical applications across various domains. By refining these tasks, NLP models achieve a nuanced understanding of language, making sentence-level analysis crucial for developing robust NLP applications.

### 7.21.2 Token-Level Tasks

Token-level tasks in Natural Language Processing (NLP) focus on **processing and understanding individual words, or tokens, within a sentence**. These tasks often serve as **foundational components** in larger NLP applications and are essential for **linguistic analysis**. Common token-level tasks include Named Entity Recognition, Part-of-Speech Tagging, and Question Answering.

- **Named Entity Recognition (NER):**

Named Entity Recognition is the task of **identifying and classifying named entities** within a text into predefined categories such as **person, organization, location, date**, and **numerical values**. NER is **vital** for information extraction in applications like search engines, digital assistants, and knowledge base construction. By tagging named entities, NER helps transform unstructured text into structured information, making it easier for systems to interpret text and identify key components. Deep

learning models, particularly **sequence-to-sequence architectures** and transformers, have proven effective for NER due to their ability to capture contextual relationships in sentences.

- **Part-of-Speech (POS) Tagging:**

Part-of-Speech tagging is the task of **assigning parts of speech** (such as **noun**, **verb**, **adjective**, etc.) to each token in a sentence. POS tagging is an essential step in **syntactic parsing** and is used in a variety of downstream tasks like sentiment analysis and language translation. Traditional POS tagging approaches include rule-based and statistical models, while modern systems utilize neural networks and deep learning models to achieve higher accuracy, especially in ambiguous linguistic contexts. POS tagging improves the interpretability of text data and is foundational for understanding sentence structure in NLP systems.

- **Question Answering (QA):**

Question Answering at the token level involves **identifying relevant spans of text, or tokens, within a given document to answer a specific question**. This task is common in applications such as chatbots, virtual assistants, and customer service automation, where the goal is to locate **precise answers** within a text based on user queries. QA models, particularly those based on transformer architectures (like BERT and GPT), have significantly advanced token-level question answering by leveraging self-attention mechanisms to locate contextually relevant information in text. In this way, QA systems can extract concise answers, often in real-time, improving the overall efficiency of information retrieval applications.

Token-level tasks are **critical** for breaking down text data into **interpretable units**, enabling NLP models to deliver detailed linguistic insights and support a wide range of applications across domains.

### 7.21.3 Downstream Tasks

In Natural Language Processing (NLP), downstream tasks refer to **real-world applications** where pre-trained language models are **fine-tuned to perform specific, practical tasks**. These applications leverage the **linguistic** and **contextual** understanding gained during **pre-training** to handle complex language tasks in various domains. Common downstream tasks include machine translation, text summarization, and chatbot development.

- **Machine Translation:**

Machine translation is the task of automatically **translating text from one language to another**. Pre-trained language models, especially transformer-based architectures, have dramatically improved machine translation by capturing deep contextual relationships between words and phrases across languages. Models like BERT, GPT, and more specialized transformer-based architectures such as MarianMT and mBERT,

can be **fine-tuned** for high-quality translations in many language pairs. This capability is essential in global communication, enabling businesses, governments, and individuals to overcome language barriers in real time.

- **Text Summarization:**

Text summarization involves **condensing a longer document into a shorter, coherent version that retains the essential information**. There are **two main types** of summarization: **extractive**, where the model selects important phrases or sentences from the original text, and **abstractive**, where it generates novel sentences that capture the text's main ideas. Pre-trained models fine-tuned for summarization tasks, such as BART and T5, are able to perform well on abstractive summarization tasks, enabling applications in news aggregation, report generation, and academic research.

- **Chatbots and Conversational AI:**

Chatbots are AI systems that **simulate human-like conversations with users**. By applying **pre-trained models**, chatbots can handle a wide range of user inquiries, providing responses based on conversational context. Fine-tuning transformer models like GPT-3 or BERT for chatbot applications helps generate more natural, context-aware responses. This has practical implications for customer support, personal assistants, and entertainment. Chatbots utilizing fine-tuned models are capable of understanding and responding to user inputs more effectively than rule-based or simpler neural network models, offering a more interactive and satisfying user experience.

These downstream tasks demonstrate the **flexibility and power of pre-trained NLP models**, enabling advanced applications that can process and understand human language at a high level. By **fine-tuning** these models for specific applications, researchers and developers can harness their broad linguistic capabilities for targeted, practical uses across diverse industries.

# Chapter 8

## Code Engineering

### 8.1 Datasets

This project utilizes three key datasets for training and testing the model to detect AI-generated text. These datasets have been curated to provide a variety of content for distinguishing between human-written and AI-generated text. Below are the descriptions of the datasets:

#### 8.1.1 Detect AI Generated Text

The **Detect AI Generated Text**<sup>1</sup> dataset, sourced from a Kaggle competition, is designed to detect whether an essay was generated by a large language model (LLM) or written by a student. This dataset serves as a challenging benchmark for distinguishing AI-generated content from human-written text, particularly in the context of essay writing.

**Dataset Description** The competition dataset comprises approximately 10,000 essays, which include both student-written and LLM-generated texts. The primary objective of the competition is to develop models capable of accurately classifying essays based on their origin.

- All essays were written in response to one of seven essay prompts. These prompts required students to read one or more source texts and write a response. The same source material may have been provided to LLMs when generating their essays.
- The training set is composed primarily of student-written essays, with only a few AI-generated essays included as examples. These examples serve as references to help build models for detecting AI-generated text.
- Essays from two specific prompts are included in the training set, while essays from the remaining prompts are reserved for the hidden test set.

---

<sup>1</sup><https://www.kaggle.com/competitions/llm-detect-ai-generated-text/data>

**Dataset Format** The dataset is structured into several CSV files, which are described below:

- `train_essays.csv` and `test_essays.csv`:
  - **id**: A unique identifier for each essay.
  - **prompt\_id**: Identifies the essay prompt the text was written in response to.
  - **text**: The full text of the essay.
  - **generated**: A binary label indicating whether the essay was written by a student (0) or generated by an LLM (1). This field is only available in the training set and is absent in the test set.
- `train_prompts.csv`:
  - **prompt\_id**: A unique identifier for each essay prompt.
  - **prompt\_name**: The title of the prompt.
  - **instructions**: The instructions provided to students for writing their essays.
  - **source\_text**: The text of the article(s) that essays were written in response to, formatted in Markdown. The source text includes numbered paragraphs for easy reference, and titles or author names where available.
- `sample_submission.csv`: A template for submission files in the correct format for the competition.

## Dataset Notes

- The test dataset used for evaluation is hidden; the `test_essays.csv` file provided in the competition dataset only contains dummy data. During the competition, this dummy data is replaced with the actual test data for scoring.
- Participants are encouraged to generate additional AI essays using various LLMs to expand the training dataset, potentially improving the model’s detection performance.

This dataset provides a diverse and challenging environment for training models to detect AI-generated text. The presence of multiple prompts, various writing styles, and a mix of student and LLM-generated essays makes it a valuable resource for exploring the differences between human and AI text.

### 8.1.2 DaiGT - Proper Train

The **DaiGT - Proper Train**<sup>2</sup> dataset is a high-quality training dataset specifically curated for the task of detecting AI-generated text. This dataset includes a diverse range of essays,

---

<sup>2</sup><https://www.kaggle.com/datasets/thedrcat/daigt-proper-train-dataset/data>

both human-written and AI-generated, to enhance the generalization capabilities of machine learning models across various writing styles and domains.

**Dataset Description** The DaiGT - Proper Train dataset comprises 33,259 unique entries with a mix of student-written and AI-generated essays. This dataset was created to address the lack of a proper training dataset for detecting AI-generated text. It includes a rich collection of essays generated using various large language models (LLMs) such as ChatGPT, Llama-70b, Falcon180b, Mistral 7B, and Claude, alongside human-written essays.

- **Sources of AI-Generated Text:**

- Text generated using ChatGPT by MOTH <sup>3</sup>
- Persuade corpus contributed by Nicholas Broad <sup>4</sup>
- Text generated using Llama-70b and Falcon180b by Nicholas Broad <sup>5</sup>
- Text generated with ChatGPT by Radek <sup>6</sup>
- Claude-generated essays contributed by Darragh Dog <sup>7</sup>
- The dataset includes a random 10-fold split stratified by the source dataset to facilitate model validation.
- Each entry may include metadata such as `EssayID` and the generation prompt, if available.

**Dataset Format** The dataset is provided in a CSV file with the following columns:

- **text:** The essay text itself.
- **label:** Binary label indicating whether the essay was AI-generated (1) or human-written (0).
- **source:** The source from which the essay originated.
- **fold:** A random 5-fold split for validation.

**Dataset Notes** This dataset was designed to provide a robust foundation for training models to distinguish between AI-generated and human-written text. By including content from multiple LLMs and diverse prompts, it aims to improve the robustness of classification models in real-world scenarios. The inclusion of various LLM-generated essays ensures that models can generalize well to different AI writing styles.

---

<sup>3</sup><https://www.kaggle.com/datasets/alejopaullier/daitg-external-dataset>

<sup>4</sup><https://www.kaggle.com/datasets/nbroad/persuade-corpus-2/>

<sup>5</sup><https://www.kaggle.com/datasets/nbroad/daitg-data-llama-70b-and-falcon180b>

<sup>6</sup><https://www.kaggle.com/datasets/radek1/llm-generated-essays>

<sup>7</sup><https://www.kaggle.com/datasets/darraghdog/hello-claude-1000-essays-from-anthropic>

### 8.1.3 LLM - Detect AI Generated Text Dataset

The **LLM - Detect AI Generated Text Dataset**<sup>8</sup> is designed for training machine learning models to accurately classify whether an essay was written by a student or generated by a large language model (LLM). This dataset provides a rich mix of human-written and AI-generated essays, making it an ideal resource for developing robust detection models.

**Dataset Description** This dataset contains over 28,000 essays, which include both student-written and AI-generated texts. The primary objective is to create models capable of distinguishing between essays authored by humans and those generated by LLMs.

- The dataset is a balanced collection of essays, which includes both student-written and machine-generated content. This mix ensures that models are exposed to diverse writing styles and can generalize effectively to real-world scenarios.
- Each essay was labeled to indicate whether it was human-written (label = 0) or AI-generated (label = 1). These labels serve as the ground truth for training and evaluating classification models.
- The dataset focuses on a variety of essay prompts, which helps in capturing variations in writing patterns between human and AI-generated content.

**Dataset Format** The dataset is structured into a single CSV file with the following columns:

- **text:** The full text of the essay.
- **generated:** A binary label indicating whether the essay was written by a student (0) or generated by an LLM (1).

#### Dataset Notes

- The dataset was curated to include a balanced mix of human and AI-generated essays to improve model performance in distinguishing between the two.
- This dataset is particularly useful for developing and fine-tuning models aimed at detecting AI-generated content, which is increasingly relevant in educational and content verification contexts.
- The simplicity of the dataset format allows for easy integration into machine learning pipelines, enabling researchers to focus on feature engineering and model development.

This dataset serves as a valuable resource for training models that can accurately differentiate between human-written and AI-generated essays, providing a solid foundation for building tools that ensure the integrity of student submissions and other text-based assessments.

---

<sup>8</sup><https://www.kaggle.com/datasets/sunilthite/llm-detect-ai-generated-text-dataset>

#### 8.1.4 Overview

Collectively, the three datasets provide a comprehensive foundation for developing and evaluating models aimed at detecting AI-generated text. The **Detect AI Generated Text** dataset offers a diverse range of text samples from various domains, challenging models to classify text based on its origin. The **DaiGT - Proper Train** dataset focuses on well-structured text examples, ensuring that models are trained on properly formatted content. Finally, the **LLM - Detect AI Generated Text Dataset** centers on outputs generated by large language models, enabling models to learn the nuances of AI-generated text. Together, these datasets enable the development of robust models capable of accurately distinguishing between human-written and AI-generated content across different contexts and writing styles.

## 8.2 Data Preprocessing

In this project, we aimed to accurately classify whether an essay was written by a student or generated by a large language model (LLM). To achieve this, we utilized three distinct datasets, each containing essays with labels indicating if the text was human-written or AI-generated. The data preprocessing steps were crucial in preparing the input for training machine learning models.

### 8.2.1 Datasets

Using the datasets that we have described in the last sections, we conduct several cleaning methods for preprocessing the text to make them cleaner to some extent.

#### Data Collection and Merging

To build a robust model for detecting AI-generated text, we utilized and consolidated three distinct datasets: **Detect AI Generated Text**, **DaiGT - Proper Train**, and **LLM - Detect AI Generated Text**. Each dataset contained a mix of human-written and AI-generated essays but varied in structure and format. Therefore, it was crucial to standardize and merge these datasets into a single cohesive dataset for efficient training and evaluation.

##### 8.2.1.1 Data Collection Process

The datasets were sourced from Kaggle and loaded into separate pandas DataFrames. Each dataset contained essays along with their corresponding labels indicating whether the text was human-written (label = 0) or AI-generated (label = 1). The data collection process involved downloading the datasets and loading them into our Python environment:

Listing 8.1: Loading Datasets

```
import pandas as pd

# Load the datasets into DataFrames
df_comp_essay = pd.read_csv('competition_essay.csv')
df_daigt = pd.read_csv('daigt_proper_train.csv')
df_llm_detect = pd.read_csv('llm_detect_dataset.csv')

# Display basic information about one of the datasets
print(df_comp_essay.info())
df_comp_essay.head()
```

##### 8.2.1.2 Data Preprocessing and Merging

The original datasets varied in terms of columns and metadata. To streamline the data and prepare it for training, we standardized each dataset to contain only two columns: `text`

(the essay content) and `label` (0 for human-written, 1 for AI-generated). This simplification helped reduce noise and ensured consistency across datasets.

The following steps were followed to preprocess and merge the datasets:

1. Loaded each dataset into a pandas DataFrame.
2. Retained only the relevant columns: `text` and `label`.
3. Standardized the label format to binary values (0 for human-written, 1 for AI-generated).
4. Concatenated the datasets into a single DataFrame.
5. Removed duplicates to ensure the integrity of the consolidated dataset.

The code snippet below demonstrates how we merged the datasets:

Listing 8.2: Merging Datasets

```
# Retain only necessary columns
df_comp_essay = df_comp_essay[['text', 'label']]
df_daigt = df_daigt[['text', 'label']]
df_llm_detect = df_llm_detect[['text', 'label']]

# Concatenate datasets into one DataFrame
df_merged = pd.concat([df_comp_essay, df_daigt, df_llm_detect],
    ignore_index=True)

# Remove duplicate entries
df_merged.drop_duplicates(subset=['text'], inplace=True)

# Display the shape of the final dataset
print(f"Final dataset shape: {df_merged.shape}")
```

### 8.2.1.3 Publishing the Merged Dataset on Kaggle

To facilitate ease of use and accessibility, we published the final merged dataset on Kaggle. Furthermore, we also push the word2vec corpus into Kaggle too, it may help it later in the engineering step when we train on Kaggle. This allows others to easily download and utilize the dataset for their own experiments and model training. The consolidated dataset can be accessed using the following links:

- Dataset: `final_dataset_v1_afternb1.csv`
- Word2Vec file: `output_w2v.txt`

#### 8.2.1.4 Final Remarks

By adopting a systematic approach to data collection, preprocessing, and merging, we were able to create a unified and robust dataset that is well-suited for training models to detect AI-generated text. The consolidated dataset provides a diverse range of writing styles and sources, enhancing the model's ability to generalize across different contexts. The availability of this dataset on Kaggle further facilitates research and experimentation in the domain of AI text detection.

## 8.2.2 Data Cleaning and Preparation

Although we have merged the 3 datasets into a single standardized and consistent one, but there might be some unnecessary information such as URL, email, phone number, or some words that do not have meaning, etc, which can harm the time complexity of our code when we have to let our model to train on these. So, to ensure the quality of input data, we applied a series of preprocessing steps. The following Python code snippets illustrate the specific techniques used.

### 8.2.2.1 Importing Libraries and Initial Setup

We started by importing essential libraries:

Listing 8.3: Import Librairies

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import json
import re
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('stopwords')
```

### 8.2.2.2 Text Cleaning

The text was cleaned by removing URLs, special characters, and non-alphanumeric characters, as well as converting the text to lowercase:

Listing 8.4: Text Cleaning

```
def clean_text(text):
    # Step 1: Remove URLs
    text = re.sub(r'http\S+|https?:\/\/\S+\www\S+', ' ', text)

    # Step 2: Remove text in square brackets
    text = re.sub(r'\[\.*?\]', ' ', text)
```

```

7   # Step 3: Remove angle brackets
8   text = re.sub(r'<.*?>', '', text)
9
10  # Step 4: Remove newlines, tabs, carriage returns
11  text = re.sub(r'\n|\t|\r', '', text)
12
13  # Step 5: Remove words containing numbers
14  text = re.sub(r'\w*\d\w*', '', text)
15
16  # Step 6: Remove non-alphanumeric characters and make lowercase
17  text = re.sub(r'\W+', ' ', text).lower().strip()
18
19  # Tokenize the text using NLTK
20  tokens = word_tokenize(text)
21
22  return ' '.join(tokens)

```

This function was applied to the dataset as follows:

Listing 8.5: Import Librairies

```
df['processed_text'] = df['text'].apply(clean_text)
```

### 8.2.2.3 Stopword Removal

To further clean the text, we removed common English stopwords:

Listing 8.6: Stopword Removal

```

stop_words = set(stopwords.words('english'))

def remove_stop_words(text):
    words = word_tokenize(text)
    filtered_words = [word for word in words if word.lower() not in
                      stop_words]
    return ' '.join(filtered_words)

df['processed_text_swr'] = df['processed_text'].apply(remove_stop_words)

```

### 8.2.2.4 Tokenization

Tokenization was performed using the NLTK tokenizer, which breaks down text into individual words:

Listing 8.7: Tokenization

```
tokens = word_tokenize("Example text for tokenization")
print(tokens)
```

### 8.2.3 Feature Engineering

Additional features were extracted to improve the model's performance:

- **Text Length:** The number of words and characters was calculated to capture verbosity differences.
- **Prompt Features:** Included prompt-related metadata to provide context for the essays.
- **Source Metadata:** Leveraged information on the source of each text to distinguish between human-written and AI-generated content.

### 8.2.4 Data Splitting

We used a stratified 5-fold cross-validation approach to ensure balanced distribution of AI-generated and human-written texts in each fold:

Listing 8.8: Data Splitting

```
from sklearn.model_selection import StratifiedKFold  
  
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)  
for train_index, val_index in kf.split(df['processed_text'], df['label']):  
    X_train, X_val = df.iloc[train_index], df.iloc[val_index]
```

By implementing these preprocessing steps, we ensured that the data was ready for input into the machine learning models, facilitating accurate classification between human-written and AI-generated essays.

## 8.3 Analyze the Training Process of Models

### 8.3.1 Model: Logistic Regression

#### 8.3.1.1 Introduction

This report presents an analysis of the performance of a logistic regression model implemented using the `SGDClassifier` from the `scikit-learn` library. The model was trained on embedded features using Stochastic Gradient Descent (SGD) optimization with a logistic loss function, effectively making it a logistic regression classifier. The objective was to achieve high classification accuracy while avoiding overfitting.

#### 8.3.1.2 Training Configuration

The logistic regression model was trained using the following configuration:

- **Loss function:** `log_loss` (logistic regression).
- **Optimizer:** Stochastic Gradient Descent (SGD).
- **Number of epochs:** 20.
- **Batch size:** 32.
- **Class weights:** Computed using the `balanced` mode to handle class imbalance.
- **Training method:** The `partial_fit` method was used for incremental learning, allowing training on mini-batches.

#### 8.3.1.3 Training and Evaluation Results and Discussion

To evaluate the model's performance, we utilized the following metrics:

- **Accuracy:** Proportion of correctly classified samples.
- **Binary Cross-Entropy Loss:** Measures the difference between predicted probabilities and actual labels.
- **ROC AUC:** Indicates the model's ability to distinguish between classes.
- **Precision:** Ratio of true positives to the sum of true and false positives.
- **Recall:** Ratio of true positives to the sum of true positives and false negatives.
- **F1-score:** Harmonic mean of precision and recall.

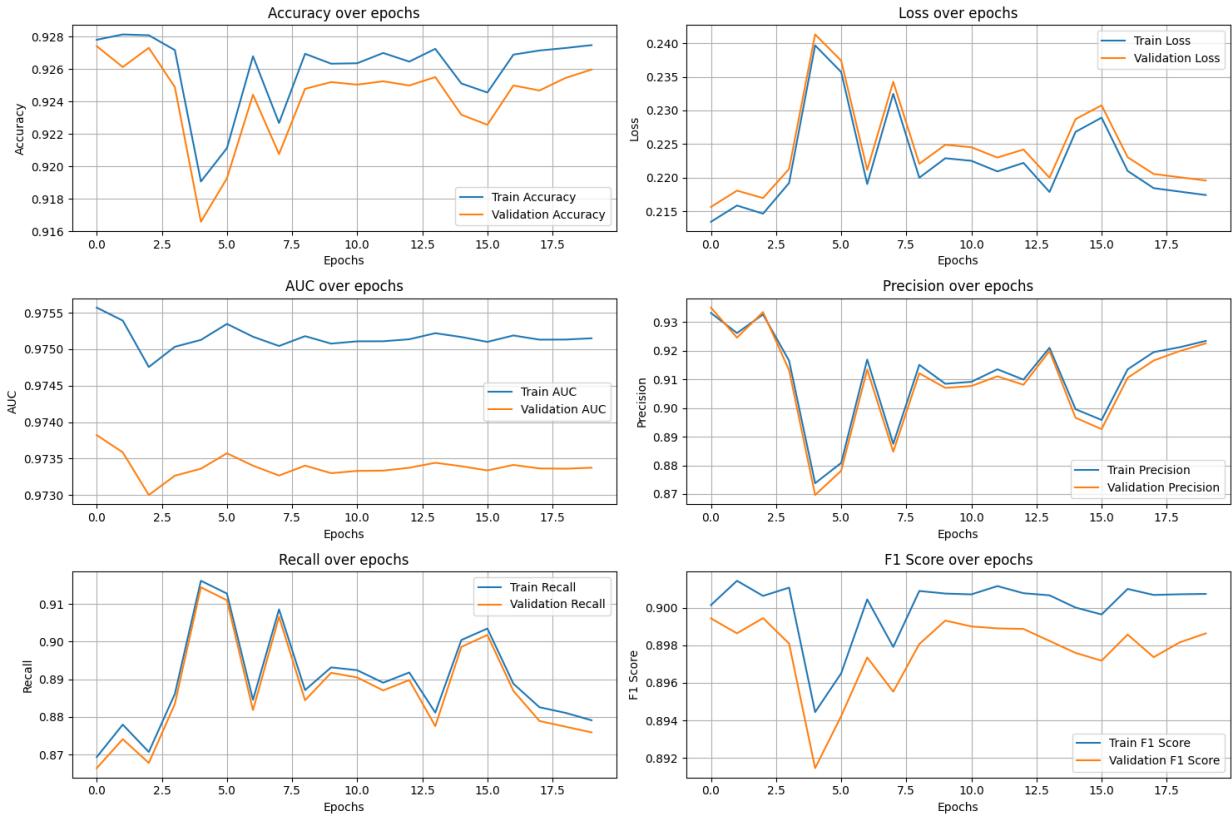


Figure 8.1: Training Metrics over Epochs

#### 8.3.1.4 Observations

- **Accuracy Analysis:** The training accuracy consistently remains higher than validation accuracy, indicating potential overfitting after several epochs. Early improvements in accuracy suggest the model quickly captures patterns in the data.
- **Loss Analysis:** Both training and validation loss decrease rapidly during the initial epochs, signifying effective learning. However, occasional spikes in validation loss towards the later epochs may indicate overfitting as the model captures noise from the training data.
- **AUC Analysis:** The ROC AUC scores remain high, showcasing the model's strong ability to distinguish between classes. The training set maintains a slightly higher AUC than the validation set.
- **Precision and Recall:** While precision and recall fluctuate, they stabilize towards the final epochs. The precision slightly decreases for the validation set, suggesting some misclassifications.
- **F1-Score:** The F1-score remains balanced, indicating that precision and recall are in harmony.

### 8.3.1.5 Test Results

After training, the model was evaluated on a separate test set, yielding the following results:

- **Test Loss:** 0.2379
- **Test Accuracy:** 92.09%
- **ROC AUC:** 0.9732
- **Precision:** 88.46%
- **Recall:** 90.56%
- **F1-score:** 89.50%

#### Discussion:

- The high ROC AUC score indicates that the model is proficient at distinguishing between positive and negative classes.
- The precision and recall values are well-balanced, leading to a high F1-score. This suggests that the model is robust and effective for binary classification.
- A slight increase in test loss relative to the validation loss suggests minor overfitting, but overall performance remains strong.

### 8.3.1.6 Conclusion

The logistic regression model implemented using the `SGDClassifier` demonstrated solid performance across multiple metrics, achieving a validation accuracy of approximately 92.5%. Despite minor overfitting, the model's precision, recall, and F1-scores indicate its reliability in distinguishing between classes. Potential future improvements include applying regularization techniques, reducing the number of epochs, or utilizing early stopping to prevent overfitting.

## 8.3.2 Model XGBoost

### 8.3.2.1 Introduction

In this analysis, we evaluate the performance of an XGBoost model implemented using the `XGBClassifier` from the `xgboost` library. The model was trained on embedded features using a learning rate of 0.01 and 100 boosting rounds. The primary objective was to achieve high classification accuracy while minimizing overfitting and ensuring good generalization.

### 8.3.2.2 Training Configuration

The XGBoost model was trained with the following configuration:

- **Learning rate:** 0.01
- **Number of estimators:** 100
- **Objective:** Binary classification (`binary:logistic`)
- **Evaluation metrics:** AUC, Log Loss, and Classification Error
- **Class weights:** Computed using the `scale_pos_weight` parameter to handle class imbalance
- **Early stopping rounds:** 20 (based on validation loss)

### 8.3.2.3 Training and Evaluation Results and Discussion

In this subsection, we analyze the performance of the XGBoost model based on the training and validation metrics recorded over 100 boosting iterations. The metrics used for monitoring the training process include AUC (Area Under the Curve), log loss, and classification error.

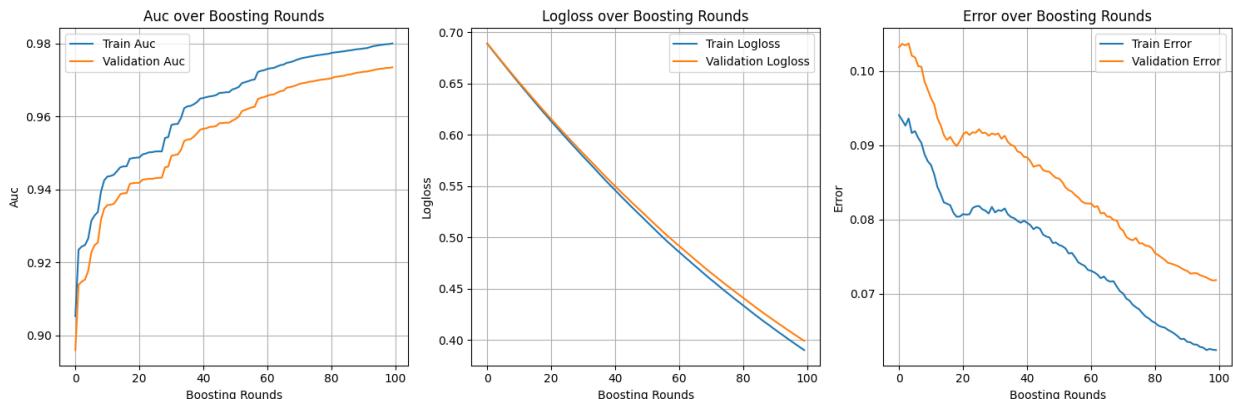


Figure 8.2: XGBoost Training Metrics over Boosting Rounds

## Observations from the Training Plots

- **AUC Analysis:** The AUC score for the training set reaches approximately 0.98, while the validation AUC stabilizes around 0.97 (Figure 8.2, left). This indicates that the model is effective at distinguishing between positive and negative classes. The rapid increase in AUC during the initial boosting rounds shows that the model quickly learns to classify the data accurately.
- **Log Loss Analysis:** The log loss for both the training and validation sets decreases steadily over the boosting iterations, as shown in the middle plot of Figure 8.2. The training log loss decreases more rapidly, reaching lower values compared to the validation log loss, which stabilizes around 0.40. This indicates that the model is optimizing well without a significant increase in overfitting.
- **Error Rate Analysis:** The classification error, depicted in the right plot of Figure 8.2, decreases continuously as the boosting rounds progress. The training error decreases more rapidly, while the validation error shows a gradual decline, eventually stabilizing. This behavior suggests that the model generalizes well to the validation set, with minimal overfitting despite the decreasing training error.
- **Overfitting Analysis:** While the training AUC slightly exceeds the validation AUC and the training error consistently drops below the validation error, the difference between these metrics is relatively small. This indicates that while some overfitting is present, it is not severe. The use of early stopping with 20 rounds likely mitigates significant overfitting.

### 8.3.2.4 Test Metrics

After training, the model was evaluated on a held-out test set to assess its generalization capabilities. The results are summarized below:

- **Test Loss:** 0.4008
- **Test Accuracy:** 92.68%
- **ROC AUC:** 0.9735
- **Precision:** 91.49%
- **Recall:** 88.55%
- **F1-score:** 90.00%

## Discussion of Model Performance

The XGBoost model demonstrates strong performance on both the validation and test sets:

- **Generalization Capability:** The test accuracy of 92.68% and ROC AUC of 0.9735 indicate robust generalization to unseen data.
- **Precision and Recall Balance:** The high precision (91.49%) and recall (88.55%) suggest a good balance between identifying true positives and minimizing false positives, leading to a high F1-score of 90.00%.
- **Overfitting Analysis:** While the training set slightly outperforms the validation and test sets, the difference is not substantial, indicating minimal overfitting.

#### 8.3.2.5 Recommendations for Improvement

Based on the observations, the following strategies could further enhance the model's performance:

- **Early Stopping:** Early stopping can prevent overfitting by halting training when the validation performance ceases to improve.
- **Hyperparameter Tuning:** Adjusting parameters such as learning rate, max depth, and subsample ratio may improve the model's performance.
- **Regularization Techniques:** Applying L1 or L2 regularization could reduce overfitting and improve generalization.
- **Feature Engineering:** Further analysis of feature importance and engineering additional features could boost model accuracy.

#### 8.3.2.6 Conclusion

The XGBoost model, trained with 100 boosting rounds and a learning rate of 0.01, achieved high accuracy, AUC, precision, recall, and F1 scores across training, validation, and test datasets. The results demonstrate the model's effectiveness for binary classification tasks on the given dataset. However, slight overfitting was observed, which could be mitigated with further optimization, such as early stopping and regularization.

### 8.3.3 Model: Random Forest

#### 8.3.3.1 Introduction

In this section, we evaluate the performance of a Random Forest model for binary classification. The model was optimized using hyperparameter tuning to maximize the ROC AUC score. Random Forest is an ensemble learning method that aggregates multiple decision trees, thereby enhancing classification accuracy and reducing the risk of overfitting.

#### 8.3.3.2 Training Configuration

The Random Forest model was trained with the following configuration:

- **Hyperparameter Tuning:** Performed using `RandomizedSearchCV`.
- **Tuned Hyperparameters:**
  - `n_estimators`: Number of trees in the forest, ranging from 50 to 500.
  - `max_depth`: Maximum depth of each tree, ranging from 1 to 20.
- **Cross-Validation:** 5-fold cross-validation was utilized.
- **Scoring Metric:** The ROC AUC score was used to evaluate model performance during hyperparameter tuning.

#### 8.3.3.3 Metrics Evaluation

The performance of the model was evaluated using the following metrics:

- **Accuracy:** Proportion of correctly classified samples.
- **ROC AUC:** Indicates the model's capability to distinguish between positive and negative classes.
- **Precision:** Ratio of true positives to the sum of true and false positives.
- **Recall:** Ratio of true positives to the sum of true positives and false negatives.
- **F1-score:** Harmonic mean of precision and recall, providing a balanced assessment.

#### 8.3.3.4 Results and Discussion

##### Learning Curve Analysis

Figures 8.3 and 8.4 display the AUC and Accuracy learning curves, respectively, as a function of the training set size.

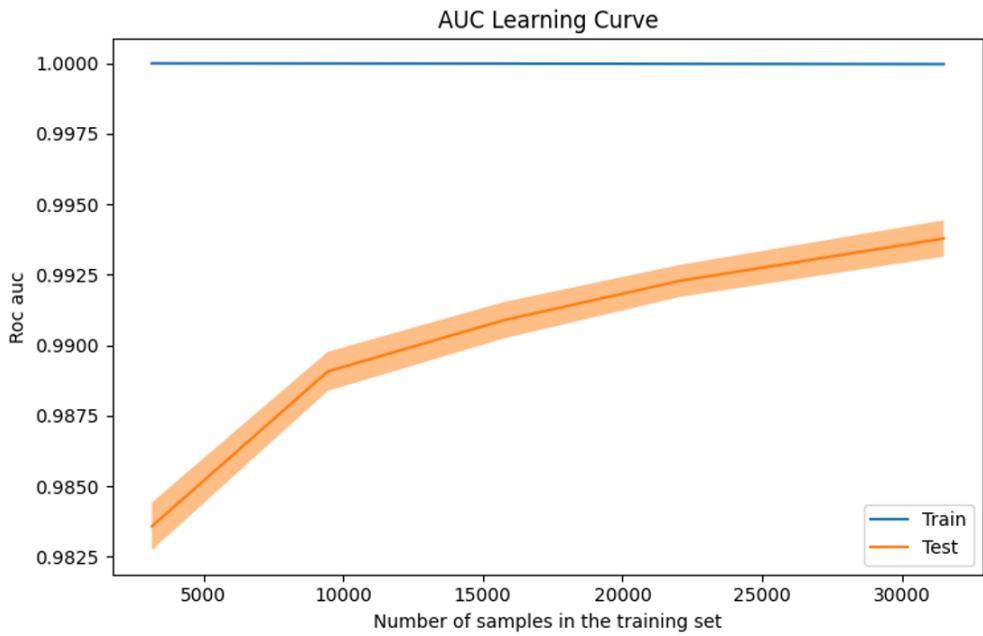


Figure 8.3: Learning Curves for ROC AUC of the Random Forest model

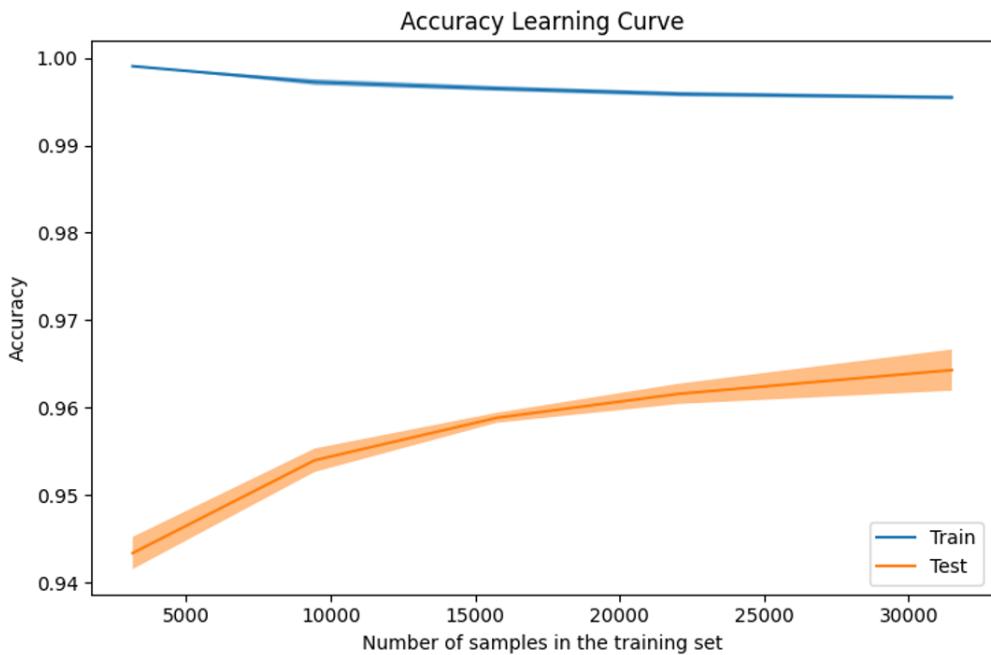


Figure 8.4: Learning Curves for Accuracy of the Random Forest model

### Observations:

- The training AUC remains consistently high, close to 1.0, demonstrating that the model effectively captures patterns in the training data.

- As the training set size increases, the test AUC gradually improves, approaching the performance of the training set, indicating that additional training data enhances generalization.
- The test accuracy also increases with more training data, reducing the variance between training and test accuracy. This reflects a reduction in overfitting with larger datasets.
- Overall, the learning curves suggest that the Random Forest model performs well on this dataset, benefiting from increased data to improve both AUC and accuracy.

## Validation Curve Analysis

Figure 8.5 shows the validation curve for the `n_estimators` parameter, which controls the number of trees in the Random Forest.

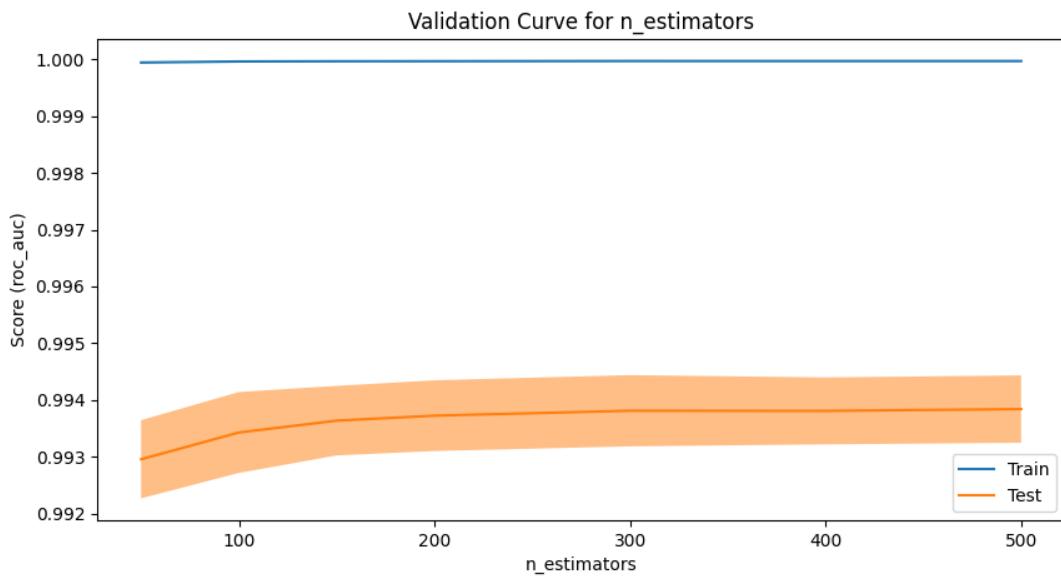


Figure 8.5: Validation Curve for `n_estimators` in the Random Forest model

## Observations:

- The training AUC remains consistently high, regardless of the value of `n_estimators`, indicating effective learning from the training data.
- The test AUC increases with more trees but plateaus around 200 trees, suggesting diminishing returns beyond this point.
- The small gap between training and test AUC scores indicates good generalization without significant overfitting.
- The optimal number of estimators appears to be between 200 and 300, balancing performance and computational efficiency.

### 8.3.3.5 Test Results

After training, the Random Forest model was evaluated on a held-out test set. The evaluation metrics are summarized as follows:

- **Test Loss:** 0.1301
- **Test Accuracy:** 96.58%
- **ROC AUC:** 0.9943
- **Precision:** 97.82%
- **Recall:** 92.88%
- **F1-score:** 95.29%

#### Discussion:

- The high ROC AUC score indicates that the model performs exceptionally well in distinguishing between classes.
- The precision and recall values are well-balanced, resulting in a high F1-score, reflecting robustness in classification.
- The low test loss and high accuracy demonstrate that the Random Forest model is well-suited for this dataset, with minimal overfitting.
- The slight increase in test loss relative to the validation set suggests that the model generalizes well to unseen data.

### 8.3.3.6 Conclusion

The Random Forest model achieved strong performance, with high accuracy, precision, recall, and ROC AUC scores on both the training and test datasets. The learning and validation curves indicate good generalization capabilities without significant overfitting. Future improvements could include fine-tuning additional hyperparameters such as `min_samples_split` and `max_features`, as well as exploring feature selection to further optimize model performance.

## 8.3.4 Model: LinearSVC

### 8.3.4.1 Introduction

This section presents an analysis of the performance of a Linear Support Vector Classifier (LinearSVC) model. The model was trained on embedded features using the `LinearSVC` implementation from the `scikit-learn` library. The objective was to achieve robust classification accuracy while minimizing overfitting using a linear kernel. LinearSVC is particularly efficient for large datasets and high-dimensional spaces.

### 8.3.4.2 Training Configuration

The LinearSVC model was trained using the following configuration:

- **Loss function:** Hinge loss.
- **Regularization parameter (C):** 1.0.
- **Class weights:** Balanced to handle class imbalance.
- **Maximum iterations:** 246.
- **Tolerance:**  $1 \times 10^{-4}$ .

### 8.3.4.3 Training and Evaluation Results and Discussion

To evaluate the model's performance, we utilized the following metrics:

- **Accuracy:** The proportion of correctly classified samples.
- **Hinge Loss:** Measures the performance of the model by penalizing misclassified samples.
- **ROC AUC:** Indicates the model's ability to distinguish between classes.
- **Precision:** Ratio of true positives to the sum of true and false positives.
- **Recall:** Ratio of true positives to the sum of true positives and false negatives.
- **F1-score:** Harmonic mean of precision and recall.

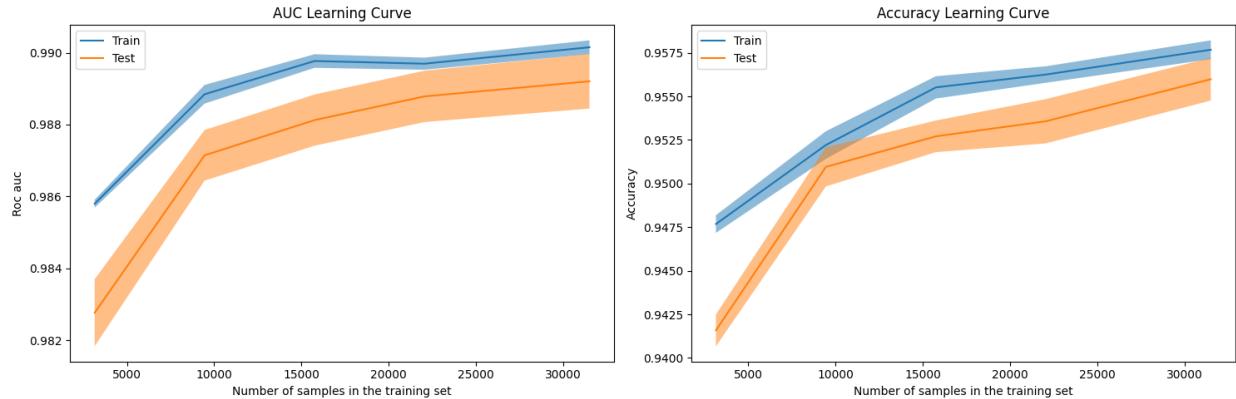


Figure 8.6: Learning Curves for AUC and Accuracy of the LinearSVC model

#### 8.3.4.4 Observations

- **Accuracy Analysis:** The learning curves in Figure 8.6 indicate that the training accuracy remains consistently higher than the test accuracy, suggesting slight overfitting. However, both curves converge as the training set size increases.
- **AUC Analysis:** The ROC AUC score remains high for both training and test sets, indicating that the model is effective at distinguishing between positive and negative classes.

#### Validation Curve Analysis

The graph in Figure 8.7 shows the validation curve for the regularization parameter  $C$ , which controls the trade-off between achieving a low training error and a low testing error.

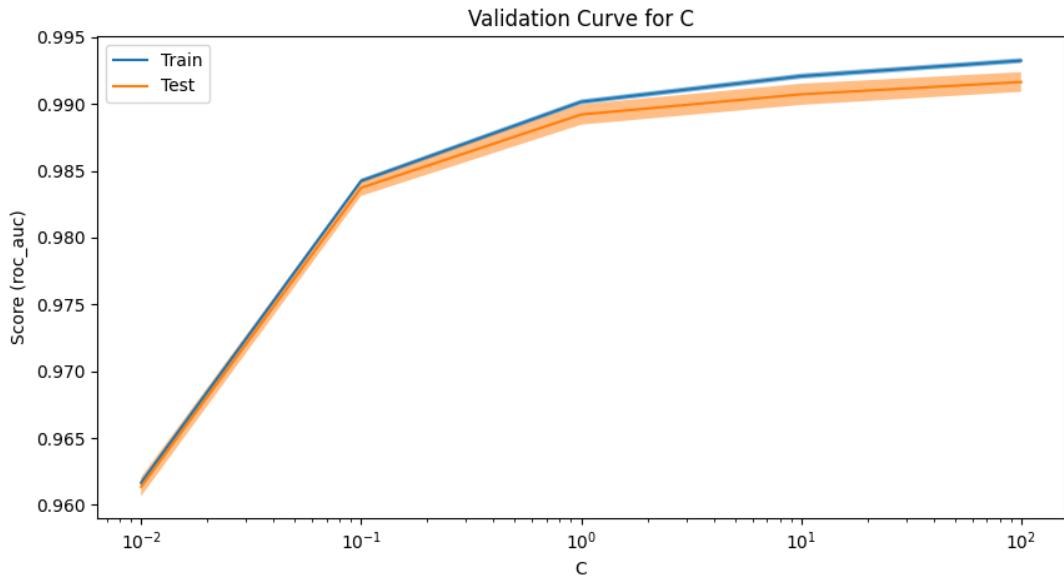


Figure 8.7: Validation Curve for  $C$  in the LinearSVC model

### Observations:

- The training AUC remains consistently high across different values of  $C$ , suggesting that the model effectively captures patterns in the training data.
- The test AUC increases with higher values of  $C$ , indicating that a slightly higher regularization parameter improves the model's ability to generalize to unseen data.
- The small gap between the training and test AUC scores indicates good generalization without significant overfitting.
- The optimal value of  $C$  for this dataset appears to be in the range of 1 to 10, balancing both performance and regularization.

#### 8.3.4.5 Test Results

After training, the LinearSVC model was evaluated on a held-out test set with the following results:

- **Test Loss (Hinge Loss):** 0.1805
- **Test Accuracy:** 95.29%
- **ROC AUC:** 0.9508
- **Precision:** 93.15%

- **Recall:** 94.27%
- **F1-score:** 93.71%

#### **Discussion:**

- The high ROC AUC score indicates that the model performs well in distinguishing between the positive and negative classes.
- The precision and recall values are well-balanced, resulting in a high F1-score, reflecting robustness in classification.
- The relatively low hinge loss and high accuracy suggest that the model generalizes well to new data, with minimal overfitting observed.

#### **8.3.4.6 Conclusion**

The LinearSVC model demonstrated strong performance with high accuracy, precision, recall, and ROC AUC scores. The learning and validation curves indicate that the model generalizes effectively while maintaining robustness. Potential improvements for future experiments include fine-tuning the regularization parameter  $C$  and exploring additional feature selection techniques to further optimize the model.

## 8.3.5 Model: Bidirectional LSTM (a type of RNN)

### 8.3.5.1 Introduction

This section presents an analysis of the performance of a Bidirectional LSTM model trained on embedded features using TensorFlow/Keras. The model leverages a bidirectional LSTM layer to capture long-term dependencies in both forward and backward directions, making it well-suited for text classification tasks. The goal was to achieve high classification accuracy while ensuring effective generalization to unseen data.

### 8.3.5.2 Training Configuration

The Bidirectional LSTM model was trained using the following configuration:

- **Text Vectorization:** Vocabulary size of 1,000 words.
- **Embedding Layer:** Output dimension of 64.
- **LSTM Layer:** 64 units in a bidirectional configuration.
- **Dense Layers:**
  - Hidden layer with 64 units and ReLU activation.
  - Output layer with sigmoid activation for binary classification.
- **Loss function:** Binary Cross-Entropy.
- **Optimizer:** Adam with a learning rate of  $1 \times 10^{-4}$ .
- **Number of epochs:** 20.
- **Batch size:** 32.
- **Metrics:** Accuracy, AUC, Precision, Recall.

### 8.3.5.3 Training and Evaluation Results and Discussion

To evaluate the model's performance, the following metrics were utilized:

- **Accuracy:** The proportion of correctly classified samples.
- **Binary Cross-Entropy Loss:** Measures the difference between predicted probabilities and actual labels.
- **ROC AUC:** Indicates the model's ability to distinguish between positive and negative classes.

- **Precision:** The ratio of true positives to the sum of true and false positives.
- **Recall:** The ratio of true positives to the sum of true positives and false negatives.
- **F1-score:** The harmonic mean of precision and recall.

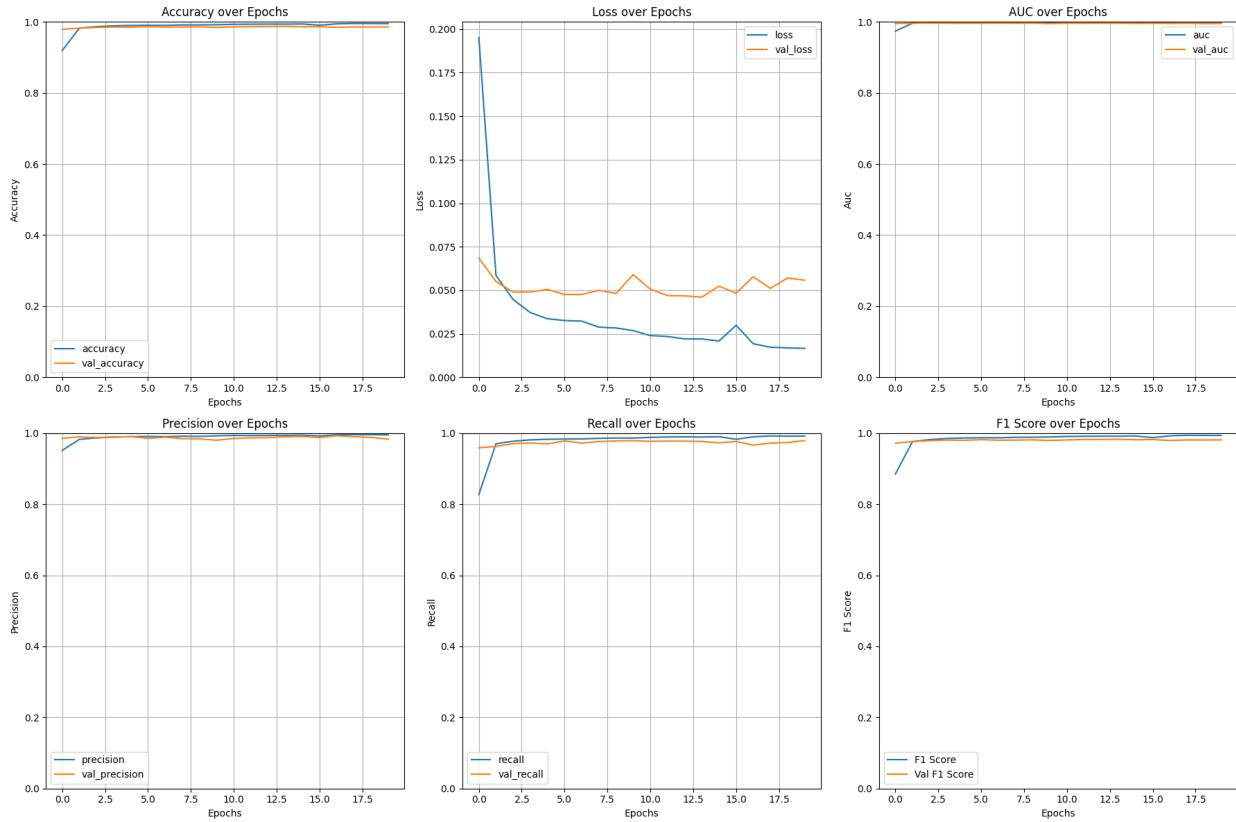


Figure 8.8: Training Metrics over Epochs for the Bidirectional LSTM RNN Model

#### 8.3.5.4 Observations

- **Accuracy Analysis:** The training accuracy reaches nearly 1.0 within the first few epochs, indicating fast learning. However, the validation accuracy stabilizes around 0.95, suggesting some potential overfitting (Figure 8.8, top left).
- **Loss Analysis:** The training loss decreases rapidly, while the validation loss fluctuates slightly but remains low, indicating effective learning but potential overfitting towards the end (Figure 8.8, top center).
- **AUC Analysis:** The AUC remains consistently high for both training and validation sets, approaching 1.0, indicating that the model is effective at distinguishing between classes (Figure 8.8, top right).

- **Precision, Recall, and F1-Score:** The precision, recall, and F1-score metrics are well-balanced, demonstrating that the model maintains a good trade-off between false positives and false negatives (Figure 8.8, bottom row).

#### 8.3.5.5 Test Results

After training, the Bidirectional LSTM model was evaluated on a held-out test set with the following results:

- **Test Loss:** 0.5599
- **Test Accuracy:** 37.21%
- **ROC AUC:** 0.9981
- **Precision:** 37.21%
- **Recall:** 100.00%
- **F1-score:** 54.24%

#### Discussion:

- The high ROC AUC score indicates that the model is highly effective at distinguishing between classes, despite the relatively low accuracy and precision.
- The perfect recall of 100% suggests that the model is successfully identifying all positive cases, but the low precision indicates a high rate of false positives, contributing to a low accuracy score.
- The F1-score of 54.24% reflects the imbalance between precision and recall, highlighting a need for further tuning to improve precision without sacrificing recall.
- The discrepancy between training/validation metrics and test results suggests potential overfitting, with the model failing to generalize effectively to unseen data.

#### 8.3.5.6 Conclusion

The Bidirectional LSTM model achieved a high ROC AUC score, indicating its effectiveness in distinguishing between positive and negative classes. However, the low test accuracy and precision highlight the need for further optimization. Potential improvements could include regularization techniques, adjusting the learning rate, and experimenting with different architectures (e.g., using GRU layers or adding dropout). Additionally, fine-tuning the vocabulary size and embedding dimensions may enhance generalization.

## 8.3.6 Model: DistilBERT

### 8.3.6.1 Introduction

This section presents an analysis of the performance of a DistilBERT model for binary text classification. The model was fine-tuned using the `transformers` library from Hugging Face. DistilBERT is a smaller, faster, and lighter version of BERT that retains much of BERT's performance while being more computationally efficient. The objective was to achieve high classification accuracy and generalization on the given dataset.

### 8.3.6.2 Training Configuration

The DistilBERT model was trained using the following configuration:

- **Pre-trained Model:** `distilbert-base-uncased`.
- **Tokenizer:** `AutoTokenizer` from the `transformers` library.
- **Learning Rate:**  $2 \times 10^{-5}$ .
- **Number of epochs:** 20.
- **Batch Size:** Determined dynamically based on available system resources.
- **Weight Decay:** 0.01.
- **Evaluation Strategy:** Evaluation was performed at the end of each epoch.
- **Dataset Split:**
  - 20% of the data was held out for testing.
  - 33% of the remaining training data was used for validation.

### 8.3.6.3 Metrics Evaluation

To assess the performance of the model, the following metrics were utilized:

- **Accuracy:** Proportion of correctly classified samples.
- **Loss (Binary Cross-Entropy):** Measures the difference between predicted probabilities and actual labels.
- **ROC AUC:** Indicates the model's ability to distinguish between classes.
- **Precision:** The ratio of true positives to the sum of true and false positives.
- **Recall:** The ratio of true positives to the sum of true positives and false negatives.

- **F1-score:** Harmonic mean of precision and recall, providing a balance between the two.

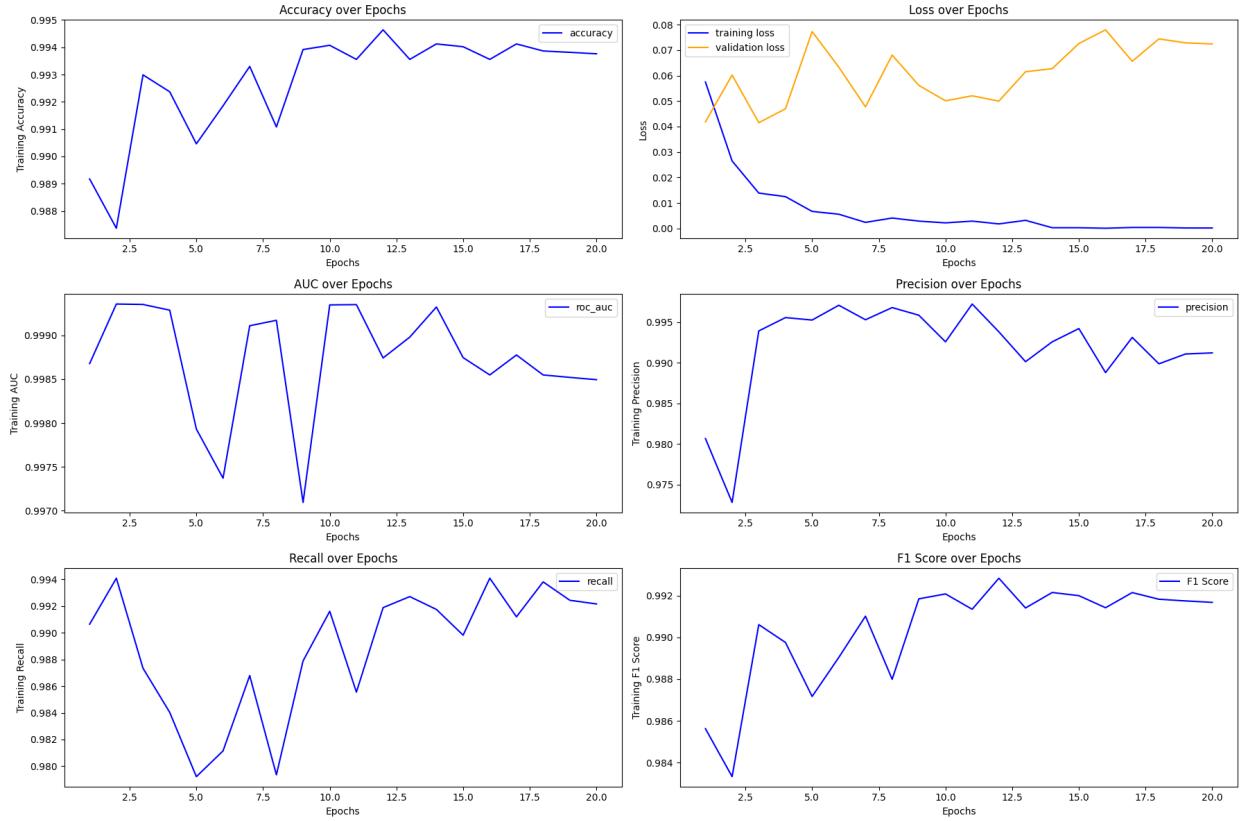


Figure 8.9: Training Metrics over Epochs for the DistilBERT Model

#### 8.3.6.4 Results and Discussion

##### Training and Validation Metrics Analysis

###### Observations:

- **Accuracy Analysis:** The accuracy fluctuated slightly throughout the epochs but showed an overall stable improvement. The training accuracy reached as high as 0.993761 by the final epoch. The validation accuracy remained high after training through 20 epochs, indicating that the model was able to generalize well across training and validation datasets, with a validation accuracy of 0.993466 at the end of training.
- **Loss Analysis:** The training loss decreased rapidly and stabilized towards the end of training, with a final value close to 0.0002. The final evaluation loss was 0.0847.
- **AUC Analysis:** The ROC AUC score remained very high, indicating the model's strong ability to distinguish between positive and negative classes. The AUC for both training and validation sets was consistently close to 1, which is a great sign of good

classification ability. The training AUC reached 0.998495, while the evaluation AUC after training over 20 epoches was slightly lower at 0.998065.

- **Precision, Recall, and F1-score:** The precision and recall values were very close to each other, indicating a balanced performance of the model across both metrics. The F1 score also remained high throughout the epochs, demonstrating that the model had a good balance between precision and recall. The final training F1 score was 0.991679, while the evaluation F1 score was 0.991239, suggesting minimal overfitting.

**Learning Curves** The learning curves show steady improvement over time. The training loss decreases smoothly, while the validation loss shows minor fluctuations, which is typical during training. Both curves indicate that the model is learning well without overfitting, as the validation loss closely tracks the training loss.

#### 8.3.6.5 Test Results

After training, the DistilBERT model was evaluated on a held-out test set. The evaluation metrics are reported as follows:

- **Test Loss:** 0.0847
- **Test Accuracy:** 0.9935
- **ROC AUC:** 0.9991
- **Precision:** 0.9891
- **Recall:** 0.9934
- **F1-score:** 0.9912

#### Discussion:

- The model performed exceptionally well across all metrics. Both the **training** and **evaluation** results show strong performance, with **training accuracy** reaching up to 0.993761, **validation accuracy** being 0.993466, and **test accuracy** at 0.9935. This high consistency across the training, validation, and test datasets indicates that the model is not overfitting and can generalize well to unseen data.
- The **training loss** steadily decreased and stabilized, and the **validation loss** followed a similar trend, with a final value of 0.0002. This suggests that the model was learning effectively and did not overfit. The **test loss** was slightly higher at 0.0847, but it is expected since the model might perform slightly worse on unseen test data compared to the validation set.

- The **ROC AUC** values were consistently high, with **training AUC** at 0.998495, **validation AUC** at 0.998065, and **test AUC** at 0.9991. This indicates the model's strong ability to distinguish between positive and negative classes, which is an essential characteristic for classification tasks.
- **Precision** and **Recall** values were well balanced throughout training, validation, and testing. The **training precision** peaked at 0.991202, while **test precision** was 0.9891. Similarly, the **training recall** reached 0.993761, while the **test recall** was 0.9934. These values suggest that the model is identifying both classes accurately without heavily favoring one over the other.
- The **F1-score** also remained high across training (0.991679), validation (0.991239), and testing (0.9912). The F1-score, being the harmonic mean of precision and recall, demonstrates that the model maintains a good balance between false positives and false negatives, making it a reliable classifier overall.
- In summary, the model's performance on the test set is highly consistent with the validation and training results. The small fluctuations in **validation loss** and **validation metrics** suggest that the model has generalized well and is not overfitting. The **test set results** further validate this, with excellent accuracy, ROC AUC, precision, recall, and F1-score. The model has shown robust performance across all datasets and metrics, making it suitable for real-world applications.

### 8.3.7 Model Comparison for AI-Generated Text Detection

In this section, we compare the performance of different machine learning models on the task of detecting AI-generated text. The models include traditional machine learning classifiers (Logistic Regression, XGBoost, Random Forest, Linear SVC), a type of Recurrent Neural Network (Bi-LSTM), and a Transformer-based model (DistilBERT). The comparison is based on key evaluation metrics, including Test Loss, Test Accuracy, ROC AUC, Precision, Recall, and F1-score.

Metric	LR	XGBoost	RF	Linear SVC	Bi-LSTM	DistilBERT
<b>Loss</b>	0.2379	0.4008	0.1301	0.1805	0.5599	<b>0.0847</b>
<b>Accuracy</b>	0.9209	0.9268	0.9658	0.9529	0.3721	<b>0.9935</b>
<b>ROC AUC</b>	0.9732	0.9735	0.9943	0.9508	0.9981	<b>0.9991</b>
<b>Precision</b>	0.8846	0.9149	0.9782	0.9315	0.3721	<b>0.9891</b>
<b>Recall</b>	0.9056	0.8855	0.9288	0.9427	1.0000	<b>0.9934</b>
<b>F1-score</b>	0.8950	0.9000	0.9529	0.9371	0.5424	<b>0.9912</b>

Table 8.1: Comparison of AI-Generated Text Detection Models

#### 8.3.7.1 Discussion

- **Logistic Regression:** This classical machine learning model performs reasonably well with an accuracy of 92.09% and a ROC AUC of 0.9732. The precision and recall are balanced, though it performs slightly worse compared to more complex models like DistilBERT and Random Forest. Logistic Regression is a simple, interpretable model that can serve as a baseline in AI-generated text detection tasks. It is computationally efficient with relatively low time complexity, making it suitable for quick deployments, especially with smaller datasets. However, it might struggle with complex data, and its performance is limited compared to more advanced models.
- **XGBoost:** XGBoost performs well with an accuracy of 92.68% and a ROC AUC of 0.9735. It outperforms Logistic Regression in terms of precision, though the recall is slightly lower. XGBoost is a powerful model that often performs well on structured data, and it's relatively fast for training on moderate-sized datasets. However, it is more complex than Logistic Regression, and its time complexity is higher due to the need for decision tree construction and boosting. Despite this, XGBoost strikes a good balance between performance and efficiency for many real-world applications.
- **Random Forest:** Random Forest achieves the highest accuracy (96.58%) among the traditional models. With a ROC AUC of 0.9943, it shows excellent discrimination between AI-generated and human-generated text. The precision (97.82%) and F1-score (95.29%) are also the highest, indicating that the model is very effective at minimizing false positives and false negatives. Random Forest is computationally expensive, especially for large datasets, as it requires constructing multiple decision trees. This model

might be slower to train than Logistic Regression and XGBoost but delivers superior performance in terms of accuracy and robustness.

- **Linear SVC:** Linear SVC performs decently with an accuracy of 95.29% and a ROC AUC of 0.9508. However, it lags behind Random Forest in terms of precision and F1-score, making it less suitable for this task compared to Random Forest or XGBoost. Linear SVC is computationally efficient, but its performance is limited compared to ensemble methods like Random Forest. It works well when the classes are linearly separable but struggles with more complex, non-linear relationships in the data.
- **Bi-LSTM:** Despite having an impressive ROC AUC of 0.9981, the Bi-LSTM shows weak performance on this task, with a very low accuracy of 37.21% and a high recall of 100%. This suggests the model is unable to differentiate well between classes, resulting in a very high number of false positives. The Bi-LSTM RNN might struggle to generalize well on this particular task and would require more tuning or a different architecture. Additionally, RNNs, especially Bi-LSTMs, have a high computational cost, especially in terms of time complexity during training, which could hinder their use in real-time applications.
- **DistilBERT:** DistilBERT achieves the best results across all metrics with a test accuracy of 99.35%, a ROC AUC of 0.9991, precision of 98.91%, recall of 99.34%, and F1-score of 99.12%. This model excels in handling complex language patterns and effectively distinguishes AI-generated text from human-generated text. DistilBERT, being a Transformer-based model, has significantly higher computational and time complexity compared to classical machine learning models, especially during both training and inference. It requires more computational resources and time, but the performance gains make it a very powerful choice for real-world applications requiring high accuracy. While the model is computationally expensive, the high performance justifies its use for detecting AI-generated text, particularly when large-scale datasets and high accuracy are important.

#### 8.3.7.2 Type I and Type II Error Considerations

The detection of AI-generated text, particularly in academic research, involves a trade-off between two types of errors: Type I errors (false positives) and Type II errors (false negatives). Understanding and balancing these errors is critical in maintaining the integrity of academic work while ensuring fairness for researchers.

- **Type I Error (False Positives):** A Type I error occurs when a model falsely identifies human-written content as AI-generated. In academic contexts, such errors can result in undue scrutiny for researchers, potentially damaging their reputation or discouraging the use of advanced tools like AI writing assistants. This could lead to unnecessary reputational risks and could harm researchers' careers or productivity. In this case, models with higher precision, such as **Random Forest** and **DistilBERT**, are favorable as they minimize false positives, ensuring that human-authored work is less likely to be misclassified as AI-generated.

- **Type II Error (False Negatives):** A Type II error occurs when AI-generated content is incorrectly classified as human-written. This can be particularly harmful in academic research, as AI-generated work could be misrepresented as original research, leading to a degradation of academic standards and integrity. **DistilBERT** excels here with its high recall rate, ensuring that most AI-generated content is correctly identified. However, it is crucial to ensure that the model does not favor recall at the cost of precision, as excessive Type II errors would undermine the detection system's trustworthiness.
- **Balancing the Errors:** The **DistilBERT** model offers the best trade-off between Type I and Type II errors, with a high precision of 98.91% and a high recall of 99.34%. This means the model performs well at both minimizing false positives and detecting AI-generated content. However, its complexity and resource requirements must be considered. On the other hand, **Random Forest** also provides good precision and recall while being computationally more efficient, making it a strong choice when computational resources are constrained or when real-time performance is critical.
- **Impact of Model Choice on Research Integrity:** Given the potential harm caused by Type I and Type II errors in academic contexts, selecting a model with the right balance of precision and recall is crucial. **DistilBERT**, with its high performance, minimizes both types of errors and is the ideal choice for ensuring the integrity of academic work. However, for cases where computational efficiency and running time are of greater concern, **Random Forest** is a solid alternative, offering a good balance between performance and resource consumption.

#### 8.3.7.3 Model Submissions to Kaggle Competition

As part of the evaluation process, five of the six models discussed earlier were submitted to the Kaggle competition titled *LLM - Detect AI Generated Text*<sup>9</sup>. The submissions included models based on Logistic Regression (LR), Recurrent Neural Network (Bi-LSTM), Random Forest (RF), XGBoost, and the Transformer-based model DistilBERT. The performance of these models was assessed using the competition's private and public leaderboards, as shown in Figure 8.10.

---

<sup>9</sup><https://www.kaggle.com/competitions/llm-detect-ai-generated-text/>

LLM - Detect AI Generated Text						Late Submission	...		
	Overview	Data	Code	Models	Discussion	Leaderboard	Rules	Team	Submissions
 <a href="#">submit_BERT - Version 3</a>							0.709609	0.842078	<input type="checkbox"/>
 <a href="#">submit_BERT - Version 2</a>							0.691199	0.796495	<input type="checkbox"/>
 <a href="#">submit_RNN - Version 2</a>							0.675486	0.765552	<input type="checkbox"/>
 <a href="#">submit_RF - Version 1</a>							0.643580	0.728436	<input type="checkbox"/>
 <a href="#">submit_XGB - Version 1</a>							0.644905	0.638830	<input type="checkbox"/>
 <a href="#">submit_LR - Version 4</a>							0.659424	0.681188	<input type="checkbox"/>

Figure 8.10: Kaggle competition submissions for detecting AI-generated text

### Description of Submissions:

- **DistilBERT (submit\_BERT):** This model achieved the public leaderboard score of 0.7965 and a private score of 0.6912 (with the only 1 epoch in the training process) and the highest among 5 models with public leaderboard score of 0.8421 and a private score of 0.7096 (with the only 1 epoch in the training process), indicating its strong generalization capability. The high scores align with the earlier evaluation metrics, where DistilBERT outperformed other models in terms of accuracy, F1-score, and ROC AUC. Despite being computationally intensive, it excelled in both public and private evaluations, making it the most reliable model for this task.
- **Bi-LSTM (submit\_RNN):** The Bi-LSTM model performed moderately well with a public score of 0.7656 and a private score of 0.6755. This result reflects its high recall rate but lower precision, as discussed earlier. The model's ability to identify AI-generated text comes at the cost of a higher false positive rate, which could explain its slightly lower private score.
- **Random Forest (submit\_RF):** Random Forest achieved respectable scores with a public leaderboard score of 0.7284 and a private score of 0.6436. This model balanced performance and computational efficiency, offering a solid alternative to more complex models like DistilBERT, especially in scenarios where computational resources are limited.
- **XGBoost (submit\_XGB):** The XGBoost model, known for its robust performance on structured data, obtained a public score of 0.6388 and a private score of 0.6449. While it performed better than Logistic Regression, it did not match the performance

of ensemble methods like Random Forest or deep learning models. Its efficiency in terms of speed and resource utilization makes it a viable option for faster deployments.

- **Logistic Regression (submit\_LR):** As a baseline model, Logistic Regression achieved a public score of 0.6812 and a private score of 0.6594. Although it lagged behind more sophisticated models, it offered reasonable performance with minimal computational overhead. This model is well-suited for scenarios requiring quick and interpretable results without the need for extensive computational resources.

#### Why Linear SVC Was Not Submitted:

The Linear Support Vector Classifier (SVC) model was not included in the final submissions to the Kaggle competition due to its inherent limitation in generating probability scores. The competition required predictions to be submitted as probabilities indicating the likelihood that a given text was AI-generated. However, Linear SVC is designed to provide deterministic, discrete outputs (i.e., class labels of either -1 or 1) rather than probabilistic scores. While it is possible to approximate probabilities using techniques like Platt scaling, such methods introduce additional complexity and may not provide reliable probability estimates. Given this limitation and the need for accurate probability outputs in the competition, the Linear SVC model was deemed unsuitable for submission.

##### 8.3.7.4 Evaluation

The results from the Kaggle competition align with the model evaluations conducted earlier. The Transformer-based DistilBERT model consistently outperformed other models, demonstrating its superiority in handling complex text patterns and effectively distinguishing between AI-generated and human-written text. However, its computational intensity may limit its deployment in resource-constrained environments.

The Random Forest and Bi-LSTM models also performed well, striking a balance between performance and computational efficiency. While the Bi-LSTM model showed strong recall, it struggled with precision, leading to a lower private score. In contrast, Random Forest demonstrated robust overall performance with fewer computational demands, making it a suitable choice when resources are limited.

Traditional models like Logistic Regression and XGBoost provided competitive performance but were outperformed by deep learning models, particularly in terms of recall and F1-score. Nevertheless, their lower computational cost makes them attractive options for applications where quick results are needed with limited hardware capabilities.

Overall, the Kaggle competition results highlight the trade-offs between model complexity, performance, and computational efficiency. For large-scale deployments where accuracy is critical, DistilBERT remains the best choice, while Random Forest and XGBoost offer efficient alternatives for real-time applications.

### 8.3.8 Conclusion

The results show that **DistilBERT** performs the best in detecting AI-generated text, followed by **Random Forest**. While traditional models like **Logistic Regression**, **XGBoost**, and **Linear SVC** provide competitive performance, the Transformer-based **DistilBERT** model outperforms them by a large margin, especially in terms of accuracy, F1-score, and recall. The **Bi-LSTM**, despite its high ROC AUC, does not generalize well on this task and may need further tuning.

For real-world applications, **DistilBERT** is the most reliable choice given its strong generalization ability and ability to balance Type I and Type II errors effectively. However, **Random Forest** might be preferred when computational efficiency is a priority, especially for smaller datasets. **Logistic Regression** and **XGBoost**, while not as powerful as **DistilBERT**, offer reasonable performance with much lower computational cost, making them suitable for tasks where speed and simplicity are important.

In summary, the trade-off between model complexity, performance, running time, and error management must be considered when selecting the most appropriate model for detecting AI-generated text in academic applications. The key is to minimize both Type I and Type II errors to ensure the integrity of academic research while being mindful of computational constraints in real-time systems.



# Chapter 9

## Self-Reflection

### 9.1 About Our Blog

In this report, we have presented key theoretical aspects of machine learning and deep learning, aiming to provide a solid foundation for understanding these rapidly evolving fields. However, theory is only the beginning of a much larger exploration.

To further support this journey, we have developed an online blog, which can be accessed at: <https://pdz1804.github.io/MLDL/>. This derives directly from Nguyen Tan Duc's blog<sup>1</sup>, a great learning reference source recommended by our advisor, Dr. Nguyen An Khuong. This blog serves as a complementary resource for learners, providing additional explanations, examples, and insights into various topics in machine learning and deep learning. It is a dynamic platform that we intend to continually improve and expand.

### 9.2 Future Developments

Looking forward, we plan to undertake the following enhancements:

- **Expanding the Content:** Add more theoretical topics, including advanced algorithms, optimization techniques, and the theoretical underpinnings of modern neural networks.
- **Documenting Our Learning Progress:** Share our own learning journey and experiences, reflecting on how we have improved and providing insights that others can benefit from.
- **Reorganizing the Structure:** Improve the organization of content to ensure a smoother learning experience, including clearer categorization of topics and hierarchical structuring.

---

<sup>1</sup><https://dukn.github.io/MLDL/>

- **Building a Knowledge Base:** Develop the blog into a comprehensive knowledge base that can be reused and referenced for upcoming projects, making it a valuable resource for research and implementation.
- **Interactive Examples:** Incorporate interactive demonstrations and code snippets to bridge the gap between theory and practice.
- **Improved Visuals:** Add diagrams, figures, and animations to aid in the understanding of complex concepts.

By documenting our progress and improvements, we hope to inspire others on their own learning paths while ensuring the blog remains a sustainable and evolving knowledge base. This platform will not only serve as a learning resource but also as a foundation for future projects and collaborations.

### 9.3 Special Thanks

We would like to extend our deepest gratitude to our advisor, Dr. Nguyen An Khuong for his invaluable guidance throughout our journey in learning machine learning. His mentorship has not only helped us master machine learning concepts specially and complex concepts generally, but has also provided us with a broader perspective on academic and career growth.

Beyond academic guidance, Dr. Nguyen An Khuong has played a pivotal role in offering career advice, helping us navigate our professional aspirations with confidence and clarity. He have also encouraged us to build meaningful academic relationships with senior colleagues, broadening our perspectives and creating opportunities for future collaborations. His encouragement and dedication have been instrumental in shaping our academic growth and preparing us for future challenges.

# Bibliography

- [1] Abu-Mostafa, Yaser S., Malik Magdon-Ismail, and Hsuan-Tien Lin, *Learning from Data*. Vol. 4. New York, NY, USA: AMLBook, 2012.
- [2] Bennett, K. P., *Robust Linear Programming Discrimination of Two Linearly Separable Sets*. Optimization Methods and Software, Vol. 1, 1992.
- [3] Bishop, Christopher M., *Pattern Recognition and Machine Learning*. Springer, 2006.
- [4] Boyd, S., and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.
- [5] Cox, David R., *The Regression Analysis of Binary Sequences*. Journal of the Royal Statistical Society. Series B (Methodological), 1958.
- [6] Cramer, Jan Salomon, *The Origins of Logistic Regression*, 2002.
- [7] Duda, Richard O., Peter E. Hart, and David G. Stork, *Pattern Classification*. John Wiley & Sons, 2012.
- [8] Friedman, Jerome H., Robert Tibshirani, and Trevor Hastie, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*.
- [9] McCulloch, W. S., and W. Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity*. The Bulletin of Mathematical Biophysics, 1943.
- [10] Nesterov, Y., *A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence  $O(1/k^2)$* . Soviet Math. Dokl., vol. 269, 1983.
- [11] Ng, Andrew, *CS229 Lecture Notes: Classification and Logistic Regression*.
- [12] Duc Nguyen, *Support Vector Machine*.
- [13] Rosenblatt, F., *The Perceptron, a Perceiving and Recognizing Automaton*. Project Para, Cornell Aeronautical Laboratory, 1957.
- [14] Rosasco, L., E. D. De Vito, A. Caponnetto, M. Piana, and A. Verri, *Are Loss Functions All the Same?* Neural Computation, 2004.
- [15] Schölkopf, B., A. Smola, R. C. Williamson, and P. L. Bartlett, *New Support Vector Algorithms*. Neural Computation, 2000.

- [16] CS231n: Convolutional Neural Networks for Visual Recognition.
- [17] CVXOPT
- [18] Wikipedia, *Hinge Loss*
- [19] Wikipedia, *Kernel Method*
- [20] LIBSVM – A Library for Support Vector Machines
- [21] Wikipedia, *Newton's Method*
- [22] Ben Frederickson, *An Interactive Tutorial on Numerical Optimization*, 2016.
- [23] Sklearn Kernel Functions
- [24] The Support Vector Machine's API from sklearn, `sklearn.svm.SVC`
- [25] Widrow, B., et al., *Adaptive "Adaline" Neuron Using Chemical "Memistors"*. Technical Report 1553-2, Stanford Electronics Labs, Stanford, CA, October 1960.
- [26] Andrew Ng, *CS229: Machine Learning (Lecture Notes)*
- [27] Tiep-Vu Huu, *Machine Learning co ban*
- [28] Wikipedia contributors. "Data set." Wikipedia, The Free Encyclopedia, [https://en.wikipedia.org/wiki/Data\\_set](https://en.wikipedia.org/wiki/Data_set).
- [29] European data portal, [data.europa.eu](http://data.europa.eu).
- [30] Various statistical sources on dataset structure and measurement.
- [31] Information on the statistical origins and software applications of datasets.
- [32] Fisher, R.A. *Iris flower data set*, 1936.
- [33] UCLA Advanced Research Computing. *Categorical data analysis datasets*.
- [34] University of Cologne. *Robust statistics datasets*.
- [35] StatLib. *Time series datasets from Chatfield's book*.
- [36] Gelman, A. *Bayesian data analysis datasets*.
- [37] GeeksforGeeks. *Voting in Machine Learning*. Available at: <https://www.geeksforgeeks.org/voting-in-machine-learning/>
- [38] SoulPage. *Ensemble Voting Explained*. Available at: <https://soulpageit.com/ai-glossary/ensemble-voting-explained/>
- [39] AIML. *Distinguish Between a Weak Learner vs Strong Learner*. Available at: <https://aiml.com/distinguish-between-a-weak-learner-vs-strong-learner/>

- [40] GeeksforGeeks. *Voting Regressor*. Available at: <https://www.geeksforgeeks.org/voting-regressor/>
- [41] GeeksforGeeks. *Voting Classifier*. Available at: <https://www.geeksforgeeks.org/voting-classifier/>
- [42] Alpaydin, E. (2020). *Introduction to Machine Learning*. MIT Press.
- [43] Hawkins, D. (1980). *Identification of Outliers*. Chapman and Hall.
- [44] Salgado, C. M., Azevedo, C., Proen  a, H., and Vieira, S. M. *Noise Versus Outliers*. Open Learning Library (MIT).
- [45] Qualtrics. *Sampling Methods, Experience Management Research*. Available at: <https://www.qualtrics.com/en-au/experience-management/research/sampling-methods/>.
- [46] BYJU's. *Sampling Methods. Maths Resource*. Available at: <https://byjus.com/math/sampling-methods/>.
- [47] Alpaydin, E. (2020). *Introduction to Machine Learning*. MIT Press.
- [48] Kearns, M., and Valiant, L. *On the learnability of Boolean formulae*, 1988.
- [49] Schapire, R. E. *The strength of weak learnability*, 1990.
- [50] Freund, Y., and Schapire, R. E. *A decision-theoretic generalization of on-line learning and an application to boosting*, 1995.
- [51] Y. Freund and R. E. Schapire, *Experiments with a new boosting algorithm*, Proceedings of the Thirteenth International Conference on International Conference on Machine Learning, 1996.
- [52] J. H. Friedman, *Greedy function approximation: A gradient boosting machine*, Annals of Statistics, Vol. 29, No. 5, 2001.
- [53] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
- [54] C. Olah, *Understanding LSTMs*, 2015.
- [55] I. Sutskever, O. Vinyals, and Q. V. Le, *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation*, *Proceedings of NeurIPS*, 2014.
- [56] M. Nielsen, *Neural Networks and Deep Learning*, 2015.
- [57] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data*. Vol. 4. AMLBook, 2012.
- [58] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.
- [59] R. Hecht-Nielsen, *Theory of the backpropagation neural network*, IEEE International Conference on Neural Networks, 1989.

- [60] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, Nature, Vol. 521, No. 7553, 2015.
- [61] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet classification with deep convolutional neural networks*, Advances in Neural Information Processing Systems, 2012.
- [62] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors*, Nature, Vol. 323, No. 6088, 1986.
- [63] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, Vol. 9, No. 8, 1997.
- [64] K. Cho, et al., *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation*, EMNLP, 2014.
- [65] D. J. C. MacKay, *Bayesian model comparison*, Machine Learning, Vol. 35, No. 1, 1999.
- [66] D. P. Kingma and M. Welling, *Auto-Encoding Variational Bayes*, ICLR, 2014.
- [67] A. Vaswani, et al., *Attention Is All You Need*, Advances in Neural Information Processing Systems, 2017.
- [68] D. Bahdanau, K. Cho, and Y. Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*, arXiv preprint arXiv:1409.0473, 2014.
- [69] M.-T. Luong, H. Pham, and C. D. Manning, *Effective Approaches to Attention-based Neural Machine Translation*, Proceedings of EMNLP, 2015.
- [70] J. Devlin, et al., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, Proceedings of NAACL-HLT, 2019.
- [71] A. Radford, et al., *Improving Language Understanding by Generative Pre-Training*, OpenAI, 2018.
- [72] T. Brown, et al., *Language Models are Few-Shot Learners*, Advances in Neural Information Processing Systems, 2020.
- [73] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.
- [74] G. Hinton, O. Vinyals, and J. Dean, *Distilling the Knowledge in a Neural Network*, arXiv preprint arXiv:1503.02531, 2015.
- [75] E. Jang, S. Gu, and B. Poole, *Categorical Reparameterization with Gumbel-Softmax*, arXiv preprint arXiv:1611.01144, 2016.