

Organización de Datos, Apunte del Curso

Luis Argerich, FIUBA

July 2, 2016
v2.0

Contents

1 Fundamentos	1
1.1 Introducción	1
1.1.1 Hacer una pregunta interesante	2
1.1.2 Conseguir los datos necesarios	8
1.1.3 Explorar los datos	12
1.1.4 Aplicar a los datos el algoritmo necesario	16
1.1.5 Comunicar los resultados	16
1.2 Formatos de Datos	16
1.2.1 Data Frames	16
1.2.2 Texto	19
1.2.3 Datos Matriciales	21
1.2.4 Imágenes	22
1.3 Niveles de Almacenamiento	22
1.3.1 Datos en Memoria	23
1.3.2 Datos en Disco	23
1.3.3 Datos en un Cluster	24
1.4 La Ley de Zipf y las Leyes de Potencias	24
1.4.1 Pareto, Zipf y Power-Laws	26
1.4.2 Leyes de Potencias	29
1.4.3 Propiedades de las Leyes de Potencias	30
1.4.4 Origen de las leyes de potencias	32
1.4.5 Resumen	32
1.5 Algunos Elementos de Probabilidad y Estadística	33
1.5.1 El Principio de Bonferroni	33
1.5.2 La Ecuación mas peligrosa de la historia	35
1.5.3 Frecuentistas vs Bayesianos	37
1.5.4 La Única Ley sin Explicación	42
1.5.5 Skewness	43
1.6 Algoritmos Aleatorizados	44
1.6.1 Algoritmo de Fermat	46
1.6.2 Algoritmo de Miller-Rabin	46
1.6.3 Random Walks	46
1.6.4 Markov Chains	46
1.6.5 Hill Climbing	50

1.6.6	El Algoritmo Metrópolis-Hastings	51
1.6.7	Simulated Annealing	51
1.7	MCMC	53
1.8	Gibbs Sampling	55
1.9	La Unión hace la fuerza: Ensambls	55
2	Nuestro Primer Algoritmo:KNN	57
2.1	La Métrica a emplear	58
2.1.1	Distancia Minkowsky	59
2.1.2	Distancia Manhattan	59
2.1.3	Distancia de Hamming o Norma l_0	59
2.1.4	Norma l_∞	60
2.1.5	Distancia de Mahalanobis	61
2.1.6	Distancia Coseno	61
2.1.7	Distancia de Edición o Levenshtein	62
2.1.8	Distancia de Jaccard	63
2.1.9	Distancia Geodésica	63
2.1.10	Distancia entre Grafos	63
2.1.11	Distancia para atributos categóricos: VDM	64
2.1.12	Distancias Especiales	66
2.1.13	Aprendiendo la Distancia	66
2.2	Determinando la Distancia a usar en KNN	67
2.3	Determinar el valor de K	67
2.3.1	Grid Search y Random Search en KNN	70
2.4	Sensibilidad a valores fuera de escala	71
2.5	Sensibilidad a atributos anómalos	71
2.5.1	Forward Selection	71
2.5.2	Backward Selection	72
2.6	Aproximaciones para KNN	72
2.6.1	Orden del algoritmo y eficiencia	72
2.6.2	Indices Espaciales: KD-Trees	73
2.6.3	Indices Espaciales: VP-Trees	75
2.6.4	Líderes y seguidores	77
2.6.5	Aproximación con K-Means	78
2.6.6	Editing	79
2.6.7	NN vía Grafos	80
2.6.8	LSH	81
2.7	Teoría de KNN	82
2.7.1	Teorema de Cover-Hart	82
2.8	Parzen Windows	83
2.9	KNN con pesos	84
2.10	Evitando Overfitting en KNN	85
2.11	RKNN: Ensambls basados en KNN	86
2.12	Algunas conclusiones y comentarios	86
2.13	Dimensionalidad	87
2.13.1	Manifolds	87

2.13.2 Los Datos No son Random y Siempre Tienen Pocas Dimensiones	88
2.13.3 Manifold Learning y Cambios de Dimensiones	90
2.13.4 La Maldición de la dimensionalidad	90
2.13.5 El Efecto de la Dimensionalidad sobre las Distancias	92
3 Almacenamiento Distribuido	95
3.1 Almacenamiento Distribuido	95
3.2 Map Reduce	96
3.2.1 Fase de Map	97
3.2.2 Map Partitions	98
3.2.3 Fase de Reduce	98
3.2.4 Fase de Shuffle & Sort	99
3.3 Normalización	99
3.4 KNN	101
3.4.1 NN	101
3.4.2 KNN	102
3.5 Procesamiento de Textos	103
3.5.1 Wordcount	103
3.5.2 N-gramas	104
3.6 Join	106
3.6.1 Producto Cartesiano	106
3.6.2 Inner Join y Outer Join	107
3.7 Operaciones con Matrices	108
3.7.1 Multiplicación matriz-vector	109
3.7.2 Multiplicación matriz-matriz	110
3.8 Redes Sociales	112
3.8.1 Relaciones no correspondidas	113
3.8.2 Contando Triángulos en una Red Social	113
4 Complejidad, Compresión y Teoría de la Información	117
4.1 Complejidad	117
4.1.1 Complejidad de Kolmogorov	117
4.1.2 Propiedades de $K(x)$	118
4.1.3 Intractabilidad de $K(x)$	121
4.1.4 Distancia Normalizada de Compresión (NCD)	121
4.2 Teoría de la Información	122
4.2.1 Mensajes y Alfabetos	122
4.2.2 Fuente	122
4.2.3 Códigos	123
4.2.4 Entropía	124
4.2.5 Entropía Conjunta	127
4.2.6 Entropía Condicional	128
4.2.7 Información Mutua	130
4.2.8 Ejemplo	131
4.2.9 Entropía Relativa	132

4.2.10 Entropía Cruzada	133
4.3 Teoría de la Compresión de Datos	133
4.3.1 Modelar los Datos	134
4.3.2 Codificar los Datos	134
4.3.3 No Existe un Compresor que Pueda Comprimir Cualquier Archivo	134
4.4 Códigos de Huffman	135
4.4.1 Huffman Estático	135
4.4.2 Huffman Dinámico	137
4.4.3 Modelos de Orden Superior	138
4.5 Compresión Aritmética	141
4.5.1 Implementación y Renormalización	142
4.6 PPMC	143
4.6.1 Ejemplo PPMC	144
4.6.2 PPMC Aspectos claves	151
4.6.3 Descompresión	152
4.6.4 Orden Máximo para PPMC	152
4.6.5 Variantes de PPMC	153
4.6.6 PPMQ y SSE	153
4.6.7 Modelos Generativos Basados en PPMC	154
4.6.8 PPMC conclusiones	156
4.7 DMC: Dynamic Markov Compression	156
4.7.1 Clonación	157
4.8 La Familia LZ de Compresores	158
4.8.1 RLE	159
4.8.2 LZSS	160
4.8.3 DEFLATE (LzHuf)	162
4.8.4 LZW	163
4.8.5 Snappy	169
4.8.6 LZMA	171
4.9 Block Sorting	173
4.9.1 Archivos localizados y MTF	173
4.9.2 La Transformación de Burrows y Wheeler	175
4.10 PAQ	176
4.10.1 Modelos de PAQ	177
4.10.2 SSE	178
5 Hashing	179
5.1 Introducción	179
5.1.1 Funciones de Hashing Elementales	179
5.1.2 Tablas de Hash Elementales	180
5.1.3 Resolución de Colisiones	181
5.2 Implementación de Diccionarios con Hash Tables	182
5.3 Hopscotch Hashing	184
5.4 Funciones de Hashing	186
5.4.1 FNV	186

5.4.2 Jenkins	187
5.4.3 Pearson	187
5.4.4 Funciones criptográficas	188
5.5 Hashing Universal	189
5.5.1 Definición de Hashing Universal	190
5.5.2 Hashing Universal para valores atómicos (numéricos)	190
5.5.3 Hashing Universal para claves de longitud fija	191
5.5.4 Hashing Universal para claves de longitud variable	192
5.6 Cuckoo Hashing	194
5.6.1 Cuckoo Hashing With a Stash	196
5.6.2 Estructuras de Datos basadas en Cuckoo Hashing	197
5.7 Balls & Bins	198
5.8 Hashing Perfecto	199
5.8.1 Esquema FKS	199
5.8.2 Hashing Perfecto Dinámico	201
5.8.3 Hashing Perfecto y Mínimo: Hash and Displace	201
5.8.4 Hashing Perfecto y Mínimo: Algoritmo II	204
5.9 The Hashing Trick	208
5.9.1 Método de Weinberger y Hash Kernels	209
5.9.2 Filtros de Spam Personalizados	210
5.10 El Teorema de Johnson y Lindenstrauss	210
5.10.1 Primera Proyección	211
5.10.2 Segunda Proyección	212
5.10.3 Tercera Proyección	212
5.10.4 Por qué funciona el Teorema de JL	212
5.10.5 The Hashing Trick y el Teorema de Johnson Lindenstrauss	216
6 LSH	219
6.1 LSH para la distancia de Jaccard	220
6.2 Amplificación de funciones LSH	224
6.3 La función LSH en la práctica	228
6.4 Familias de Funciones LSH	231
6.5 LSH para la distancia angular	233
6.5.1 El método de los hiperplanos	233
6.5.2 El Método de los Polítopos	234
6.6 LSH para la distancia euclideana	236
6.6.1 El Método de las Proyecciones Aleatorias	237
6.6.2 Métodos basados en K-Means: KLSH	238
6.6.3 Métodos basados en K-Means: K-Means jerárquico	239
6.7 LSH para la distancia Hamming	240
6.8 Multi-Probe LSH	240
6.8.1 Multi-Probing para la distancia euclideana, método de proyecciones aleatorias	241
6.8.2 Multi-Probing para la distancia euclideana, métodos basados en K-Means	241

6.8.3	Multi-Probing para la distancia angular, método de proyecciones aleatorias	241
6.8.4	Multi-Probing para la distancia angular, método de los polítopos	241
6.8.5	Multi-Probing para la distancia de Jaccard	241
6.9	LSH Conclusiones	242
6.10	LSH Para Máxima semejanza	243
6.10.1	Usando la Longitud	243
6.10.2	Usando Prefijos	244
6.10.3	Prefijos y Posiciones	244
6.10.4	Prefijos, Posiciones y Longitud	245
6.11	LSH forests	247
7	Patrones Frecuentes	249
7.1	A-Priori	250
7.1.1	Generación de Candidatos Intermedia	252
7.2	PCY	253
7.3	Algoritmos Aleatorizados	253
7.3.1	Muestreo Aleatorio	254
7.3.2	Algoritmo SON	254
7.3.3	Algoritmo de Toivonen	258
7.4	Reglas de Asociación	259
7.4.1	Soporte y Confianza en Reglas de Asociación	260
7.4.2	Limitaciones del Soporte y Confianza para encontrar asociaciones interesantes	260
7.4.3	Lift	261
7.4.4	Métricas invariantes a los valores nulos	262
8	Reducción de Dimensiones	263
8.1	Introducción	263
8.1.1	Para Combatir la Maldición de la Dimensionalidad	263
8.1.2	Por motivos de performance	263
8.1.3	Como un mecanismo de Feature Engineering	264
8.1.4	Para visualizar los datos	264
8.2	SVD	264
8.2.1	Interpretación algebraica de la SVD (muy importante!)	267
8.2.2	Aproximación de Rango k	267
8.2.3	Usando la SVD para aproximar la dimensionalidad intrínseca de los datos	270
8.2.4	Reducción de dimensiones usando la SVD	271
8.2.5	Compresión de Imágenes usando la SVD	271
8.2.6	Eigenfaces	273
8.2.7	SVD en la práctica	275
8.2.8	Agregando nuevos datos en el nuevo espacio	275
8.3	PCA: Principal Component Analysis	276
8.3.1	PCA=SVD	278

8.3.2	Conclusión	279
8.4	MDS: Multidimensional Scaling	279
8.4.1	Perceptual Mapping	283
8.5	ISOMAP	283
8.5.1	Construcción del Grafo	283
8.5.2	Cálculo de Distancias	284
8.5.3	MDS	284
8.6	Laplacian Eigenmaps	285
8.6.1	Creación del Grafo	285
8.6.2	Construcción del Laplaciano	286
8.6.3	Descomposición Espectral del Laplaciano	286
8.6.4	Teoría de la descomposición espectral	286
8.6.5	Ejemplos	288
8.7	TSNE	290
8.8	Teorema Fundamental de la Dimensionalidad	292
9	Information Retrieval	295
9.1	Consultas Puntuales y el Dominio de Trabajo	296
9.2	índices Invertidos	296
9.2.1	Estructura de un índice Invertido	297
9.2.2	Almacenamiento del Léxico	297
9.2.3	Almacenamiento de Punteros	299
9.2.4	Estructura General del índice	300
9.2.5	Almacenamiento de Punteros -Avanzado-	301
9.2.6	Construcción del índice Invertido	305
9.2.7	Qué Indexar	306
9.3	Consultas de Frases y Proximidad	307
9.4	Signature Files	309
9.4.1	Consultas	309
9.5	Bitmaps	310
9.5.1	Compresión de Bitmaps	311
9.6	Consultas Ranqueadas	312
9.6.1	BOW y Productos Internos	312
9.6.2	TF-IDF	313
9.7	Indexación Semántica Latente	317
9.8	Evaluación de Consultas	318
9.9	Consultas Probabilísticas	320
9.10	Procesamiento de Lenguaje Natural	320
9.10.1	Modelo de N-Gramas	320
9.10.2	Calculando Probabilidades en el modelo de N-Gramas . .	322
9.10.3	Maximum Likelihood (Máxima semejanza)	322
9.10.4	El Problema de las Probabilidades Nulas	324
9.10.5	Smoothing; Corrección de Laplace	324
9.10.6	Interpolación y Backoff	325
9.10.7	Good Turing Smoothing	326
9.10.8	Algoritmo de Kneser-Ney	327

9.11 Learning to Rank	329
9.11.1 Algoritmos Pointwise	330
9.11.2 Algoritmos Pairwise y Listwise	332
10 Visualización	335
10.1 Principios Básicos	336
10.1.1 No usar tres dimensiones de forma injustificada	336
10.1.2 El uso del plano	336
10.1.3 Oclusión de Información	336
10.1.4 Mal uso del Plano	336
10.1.5 Capacidad de Retención	337
10.1.6 Ceguera al cambio	337
10.1.7 Uso del Color	337
10.1.8 Foco en la Funcionalidad	337
10.2 Principios de Visualización de Tufte	338
10.2.1 Excelencia	338
10.2.2 Principios	339
10.3 Scatter Plots	339
10.4 Gráficos de Barras	347
10.5 Histogramas y Plots de Densidad	351
10.6 Boxplots	355
10.7 Gráficos de Líneas	362
10.8 Gráficos de área	365
10.9 Heatmaps	368
10.10 Radar Charts	370
10.11 Coordenadas Paralelas	371
10.12 Treemaps	372
10.13 Mapas y Choropleths	373
11 Streaming	375
11.1 Reservoir Sampling	375
11.2 Momentos de un Stream	377
11.3 Flajolet-Martin y el HyperLogLog	378
11.3.1 Flajolet-Martin	378
11.3.2 HyperLogLog	379
11.4 AMS	379
11.5 DGIM	380
11.5.1 Extensión para valores enteros	383
11.6 Decaying Windows	383
11.7 Filtros de Bloom	384
11.7.1 Extensión para admitir borrado	386
11.8 Count-Min	386
11.8.1 The Heavy Hitters Problem	387
11.8.2 The Count-Min Sketch	388
11.8.3 Claves Transistorias: Bit-Marking	389
11.8.4 Teoría de Count-Min	391

12 Introducción a Machine Learning	395
12.1 Evaluación de Algoritmos de ML	396
12.1.1 Cross Validation	396
12.2 Los Problemas de ML: Overfitting y Underfitting	397
12.3 El Teorema de No Free Lunch	400
12.4 La Unión hace la fuerza: Ensamblés	402
12.4.1 Bagging	403
12.4.2 Boosting	404
12.4.3 Combinando Algoritmos Diferentes	404
12.5 Teoría de Machine Learning	408
12.5.1 Desigualdad de Hoeffding	408
12.5.2 Error de Entrenamiento	409
12.5.3 Error de Generalización	409
12.5.4 Overfitting y Underfitting	409
12.5.5 El Problema de Clasificación	409
12.5.6 El Espacio de Hipótesis	410
12.5.7 Minimización Empírica del Riesgo en \mathcal{H}	410
12.5.8 Cuando $ \mathcal{H} $ es finito	410
12.5.9 Cuando $ \mathcal{H} $ es infinito	414
12.5.10 El Principio de la Mínima Descripción	417
12.5.11 Clasificación con Algoritmos de Compresión	419
13 Clasificación	421
13.1 árboles de decisión	421
13.1.1 ID3	421
13.1.2 C4.5	424
13.2 Random Forests	425
13.2.1 Distancia RF	426
13.3 XGBoost	427
13.3.1 Boosting Trees	427
13.3.2 Complejidad de un árbol	429
13.3.3 Optimizando la función objetivo	430
13.3.4 Calculando el Score de una Estructura	431
13.3.5 Buscando la Estructura del árbol	431
13.3.6 XGBoost en la práctica	432
13.4 Naïve Bayes	433
13.4.1 El Problema de las probabilidades cero	434
13.4.2 Bayes en acción	435
13.5 Perceptrón	435
13.5.1 Algoritmo Base	436
13.5.2 Algoritmo Mejorado	437
13.5.3 Normalización Online	438
13.5.4 Predicciones	438
13.5.5 Convergencia	438
13.5.6 Funciones de Activación	440
13.5.7 Perceptrón Multiclas	442

13.5.8 Conclusiones	443
13.6 SVM	443
13.6.1 Margen Funcional	445
13.6.2 Margen Geométrico	445
13.6.3 Support Vectors	448
13.6.4 El Problema Dual	450
13.6.5 Algoritmo SMO (Sequential Minimal Optimization) [opcional]	452
13.6.6 Soft Margin	454
13.6.7 SMO (Soft Marging)	456
13.7 The Kernel Trick	458
13.7.1 Kernel Polinómico	461
13.8 Teorema de Mercer	462
13.8.1 Kernel Gaussiano (RBF)	463
13.9 Aproximación de Nystrom	464
13.10 Kernel SVM	465
13.10.1 El efecto de C y σ	465
13.11 SVM Transductivo	468
13.12 SVM Ordinal	469
13.13 Conclusiones	469
14 Clustering	471
14.1 Clustering Jerárquico	472
14.1.1 Distancia Entre Clusters	472
14.1.2 Encontrando la Cantidad de Clusters	473
14.1.3 Dendrogramas	475
14.1.4 Performance	476
14.2 K-Means	476
14.2.1 El Problema General	477
14.2.2 El Algoritmo de Lloyd (conocido como K-Means)	478
14.2.3 K-Means++	481
14.2.4 Como buscar K	484
14.2.5 Diagramas de Voronoi	484
14.2.6 Kmeans Online	486
14.2.7 Soft K-Means	489
14.2.8 La base de Kmeans y sus aplicaciones	491
14.2.9 Reducción de Dimensiones con K-Means	494
14.2.10 Kmeans Esférico	496
14.2.11 Kernel K-Means	498
14.2.12 K-Modes	500
14.2.13 K-Means y el Problema de los Vecinos Mas Cercanos	501
14.2.14 The Inverted Multi-Index	503
14.3 Clustering Espectral	505
14.3.1 Primer paso: construir la matriz de afinidad	506
14.3.2 Segundo paso: construir la matriz Laplaciana	506

14.3.3 Tercer paso: Autovalores y Autovectores de la matriz Laplaciana	507
14.3.4 Selección de Hiper-Parámetros	509
14.3.5 Aproximaciones: KASP	510
14.4 CURE	510
14.5 DBScan	512
14.6 HDBScan	514
14.6.1 La Distancia MRD	514
14.6.2 El árbol Generador Mínimo	515
14.6.3 Clustering Jerárquico	515
14.6.4 Extraer los Clusters	517
14.7 Bioclustering	520
14.7.1 Co-clustering Espectral	521
14.7.2 LSI Co-clustering	523
14.8 NMF: Non-negative matrix factorization	524
14.8.1 Algoritmo Multiplicativo	525
14.8.2 La Base de NMF	526
15 PageRank y sus Derivados	529
15.1 Random Surfers	530
15.2 Convergencia	532
15.3 Dead Ends	534
15.4 Spider Traps y Teletransportación	535
15.5 Page Rank en la práctica	537
15.5.1 PageRank via Map Reduce	538
15.6 TopicRank	539
15.7 TrustRank	540
15.8 SimRank	542
15.8.1 Sim Rank via Montecarlo	542
15.8.2 Panther	543
15.9 VisualRank	543
15.10TextRank	545
15.11HITS	546
16 Redes Sociales	551
16.1 Propiedades de los Grafos	551
16.1.1 Propiedades Básicas	551
16.1.2 Almacenamiento del Grafo	552
16.1.3 Componentes Conexos	552
16.1.4 Diámetro	553
16.1.5 Coeficiente de Clustering	553
16.1.6 Distribución del Grado	554
16.2 Propiedades del Grafo de una Red Social	554
16.2.1 Componente Conexo	554
16.2.2 Distribución del Grado	554
16.2.3 Diámetro y Camino Mínimo Promedio	555

16.2.4	Coeficiente de Clustering	556
16.3	Modelos de Grafos	556
16.3.1	Modelo de Erdős-Renyi	557
16.3.2	Modelo de Barabasi-Albert	558
16.3.3	Modelo de Jackson-Rodgers	559
16.4	El Mundo Pequeño	559
16.4.1	Modelo de Watts Strogratz	560
16.5	La Clausura Triangular	561
16.5.1	Contando Triángulos en un Grafo	561
16.5.2	Triángulos Locales	564
16.5.3	EE Eigenplots	565
16.6	Centralidad	567
16.6.1	Grado	567
16.6.2	Betweenness	569
16.6.3	Closeness	570
16.6.4	Eigenvector Centrality y PageRank	571
16.6.5	Centralidad de Bonacich	571
16.6.6	Centralidad de la Red (Freeman)	572
16.7	Detección de Comunidades	573
16.7.1	Uniones Fuertes y Débiles	573
16.7.2	Modularidad	574
16.7.3	Algoritmo de Girwan-Newman	575
16.7.4	Clustering Espectral	576
16.7.5	Infomap	576
16.7.6	Comunidades con Solapamiento: Big Clam	580
16.8	Análisis de comunidades en Redes Sociales	582
16.9	Cascadas en Redes Sociales	585
16.9.1	Cascadas	585
16.9.2	Maximización de Influencia	587
16.10	Redes Sociales con Signo	588
16.11	Motifs	591
16.12	El Lenguaje en las Redes Sociales	593
16.13	Evolución de una Red Social	595
16.13.1	Cantidad de Nodos y Aristas	595
16.13.2	Diámetro	597
16.14	Grafos de Kronecker	598
16.14.1	Propiedades	599
16.14.2	Grafos de Kronecker Estocásticos	599
16.14.3	Estimación de la matriz generadora	600
16.14.4	Generando Redes Sociales con Grafos de Kronecker	601
17	Sistemas de Recomendaciones	603
17.1	Introducción: El modelo de descubrimiento	603
17.2	Características de un Sistema de Recomendaciones	604
17.2.1	Precisión	604
17.2.2	Serendipity	604

17.2.3 Diversidad	606
17.2.4 Otros desafíos para un sistema de recomendaciones	606
17.3 Sistemas Basados en Contenidos (Content Based)	608
17.4 Collaborative Filtering	609
17.4.1 Semejanza User User	610
17.4.2 Semejanza Item Item	611
17.4.3 Collaborative Filtering en base a Desviaciones	612
17.5 Evaluación de Sistemas de Recomendación	615
17.6 Modelos Latentes	616
17.6.1 SVD++	616
17.7 Factores Temporales	619
17.8 Factorization Machines	620
17.8.1 Expresividad de FM	621
17.8.2 Como funcionan los parámetros de una FM	622
17.8.3 Algoritmos basados en FMs	622
17.8.4 Aprendizaje de FMs	623
17.8.5 LibFM	623
17.9 El Problema del Cold-Start: Multi-Armed Bandits	624
17.9.1 Multi-Armed Bandits	624
17.9.2 Algoritmo Epsilon	625
17.9.3 Algoritmo UCB1	625
17.10 Diversidad: Optimización Submodular	625
17.10.1 Recomendaciones Personalizadas	628
17.11 Learning to Rank	629
17.12 El Sistema Completo	629
18 Análisis Topológico de Datos	631
18.1 Propiedades del Análisis Topológico	632
18.2 Los Datos como un Grafo	633
18.2.1 Complejo Vietoris-Rips	633
18.2.2 Complejo de Cech	636
18.3 Homología Persistente	636
18.3.1 Números de Betti y Barcodes	637
18.3.2 BarCodes	638
18.3.3 Otros puntos de Análisis	639
18.4 Algoritmo Mapper	640
18.4.1 Lentes Topológicos Comunes	640
18.4.2 Cubriendo los Datos	641
18.4.3 Fase de Clustering	643
18.4.4 Construcción del Grafo	644
18.4.5 Ejemplo	644
18.4.6 Ejemplo	647
18.5 Conclusiones	651

19 Metadatos	653
19.1 Dublin Core	654
19.2 RDF	655
19.2.1 El Modelo RDF	655
19.2.2 RDF mediante Grafos	656
19.2.3 Recursos Anónimos	656
19.3 Turtle	657
19.4 Ontologías	658
19.4.1 RDF-Schema (RDFs)	658
19.4.2 Otros elementos de RDFs	660
19.5 SPARQL	660

Chapter 1

Fundamentos

Sometimes beginnings aren't so
simple

The Shadow of the Day
Linkin Park

1.1 Introducción

Data Science es la rama de la computación que se encarga de entender, procesar y extraer valor a partir de los datos. Hoy en día existen innumerables fuentes de datos y herramientas para recolectar datos, desde un log de un website hasta la estructura de una Red Social pasando por bases de datos en todo tipo de formatos y pertenecientes a los mas diversos dominios.

Una forma muy popular de definir Data Science es mediante la intersección de tres áreas: Una que cubre los aspectos matemáticos y estadísticos, otra que cubre la parte computacional, es decir algoritmos y programación y finalmente un área que es propia del dominio en cuestión, es decir conocimiento sobre el tema del cual tratan los datos.

La aplicación de Data Science a un conjunto de datos puede darse de las mas diversas formas y con diversos objetivos, en general el proceso consiste en obtener a partir de los datos las respuestas que necesitamos a ciertas preguntas que resultan importantes. Podemos generalizar la mayoría de los procesos de Data-Science mediante los siguientes pasos:

1. Hacer una pregunta interesante
2. Conseguir los datos necesarios
3. Explorar los datos
4. Aplicar a los datos el algoritmo necesario

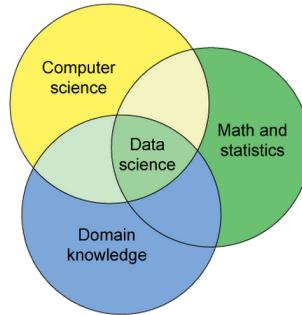


Figure 1.1: Data Science.

5. Comunicar los resultados

Cada uno de estos puntos implica una importante cantidad de trabajo, a continuación intentaremos explicar brevemente cuáles son las actividades típicas para cada paso y lo que cada una de estas actividades implica.

1.1.1 Hacer una pregunta interesante

La pregunta que queremos responder es el inicio de todo proceso de análisis de datos, es lisa y llanamente la información que queremos obtener a partir de los datos. Por ejemplo podemos tener un conjunto de datos sobre propiedades incluyendo sus características y precios y construir un modelo que nos permita tasar automáticamente una propiedad a partir de sus características. Así como este podríamos listar una enorme cantidad de problemas que podemos resolver usando Data Science. Como no hay espacio para listar todos los problemas posibles vamos a agruparlos en categorías intentando que cada una de estas categorías sea representativa de una buena cantidad de problemas. Es además una buena forma de presentar terminología que usaremos constantemente a lo largo del curso. Por favor notemos que vamos a presentar estas categorías simplemente desde el punto de vista de qué pregunta podemos hacernos en base a nuestros datos, no vamos a entrar aún en detalles ni terminología propios de cada tipo de problema, los lectores que ya tienen alguna experiencia en estos temas tendrán que esperar a que cada uno de estos temas sea desarrollado en mayor detalle.

Regresión

En un problema de regresión queremos predecir el valor de una variable numérica y continua a partir de un cierto conjunto de datos. En general contamos con un cierto set de entrenamiento en el cual conocemos el valor de la variable que

queremos predecir. El objetivo es entonces construir un modelo predictor que nos permita predecir el valor de nuestra variable de decisión a partir de datos nuevos.

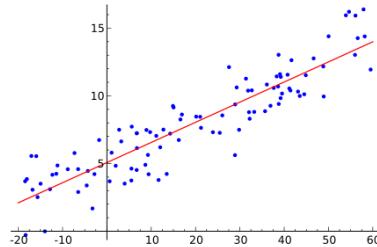


Figure 1.2: Regresión lineal.

El caso mas simple es la regresión lineal en el cual nuestro modelo es una recta, la recta que mejor ajusta a los puntos de nuestro set de entrenamiento. Los problemas de regresión pueden usarse para predecir el valor de las acciones en el mercado de valores, para estimar el costo de una propiedad, para calcular la cantidad de personas en un casino, para estimar las ganancias de un negocio que queremos abrir, etc. Las claves para identificar un problema de regresión son las siguientes:

- Queremos predecir una variable que es numérica y en general continua
- Contamos con un set de entrenamiento para el cual conocemos el valor de dicha variable

Clasificación

La clasificación automática es muy similar a la regresión pero la variable que queremos predecir no es continua sino discreta, frecuentemente tiene pocos valores posibles y en muchos casos los valores posibles son solo dos (clasificación binaria). La idea es la misma que antes, contamos con un set de entrenamiento en el cual para cada dato conocemos la clase a la cual pertenece el mismo, queremos construir un modelo que nos permita clasificar automáticamente datos nuevos cuya clase desconocemos.

Un caso típico de la clasificación binaria es el análisis de sentimiento, que podría ser una categoría en si misma dada su enorme utilidad. En un problema de análisis de sentimiento queremos saber si un cierto texto es positivo o negativo, es decir si habla bien o mal de un cierto tema. Como set de entrenamiento deberíamos contar entonces con textos para los cuales ya conocemos su sentimiento. Como podrán imaginarse esto tiene muchísimas aplicaciones como por ejemplo analizar si los reviews de un producto son buenos o malos, determinar si un usuario de una Red Social está pasando por un mal momento, medir la actitud del público en general ante determinadas noticias por los comentarios

que existen en un medio gráfico, etc. Y ésta es solo una de las muchísimas formas que puede tomar un problema de clasificación.

Otro ejemplo que estudiaremos mas adelante consiste en predecir la clase a la cual pertenece un vino en base a sus propiedades químicas, contamos como set de entrenamiento con un conjunto de vinos para los cuales conocemos su clase y el valor de ciertas propiedades químicas del mismo y en base a estos datos queremos construir un modelo que nos permita predecir la clase para vinos nuevos cuya clase desconocemos en base a sus propiedades.

Para reconocer un problema de clasificación en general hay que estar atentos a las siguientes pistas:

- Queremos determinar la clase a la que pertenece cada dato
- La clase es una variable discreta con un set de valores posibles limitado y definido
- Contamos con un set de entrenamiento para el cual conocemos los datos y a que clase pertenecen

En algunos casos existe una zona gris entre problemas de clasificación y de regresión, como ejemplo planteemos el siguiente problema: Contamos con reviews para un determinado producto, por ejemplo autos y para cada review hay un número de 1 a 5 que indica que tan positivo es el mismo, por ejemplo las típicas estrellitas con las que podemos calificar cosas en varios sitios. Este problema lo podríamos plantear como un problema de clasificación con cinco clases o bien como un problema de regresión con valores posibles del 1 al 5. La diferencia es que en un problema de regresión nuestro modelo podría predecir 3.45 para un cierto review mientras que en un problema de clasificación los valores posibles son solo 1,2,3,4 y 5. No existe una receta para resolver este tipo de situaciones; en algunos casos obtendremos mejores resultados planteando el problema como una regresión y en otros como una clasificación.

Clustering

En un problema de clustering contamos con datos que queremos dividir en grupos de forma automática, por ejemplo podemos tener artículos de noticias y querer agruparlos en categorías de forma tal que queden juntos todos los de deportes, economía, política, etc. En algunos casos la cantidad de "clusters" la debemos indicar previamente y en otros el algoritmo es capaz de determinarla por si mismo. Otro ejemplo podría ser agrupar películas automáticamente, de forma que queden juntas las que son de un mismo género. La detección de comunidades en una red social es un típico problema de clustering en donde los puntos son los usuarios y queremos agruparlos automáticamente en comunidades, de esta forma podemos descubrir grupos de usuarios que tienen un cierto interés en común aun sin saber exactamente cuál es dicho interés.

A los problemas de clustering se los suele llamar "aprendizaje no supervisado" en contraste con los problemas de regresión o clasificación que son de

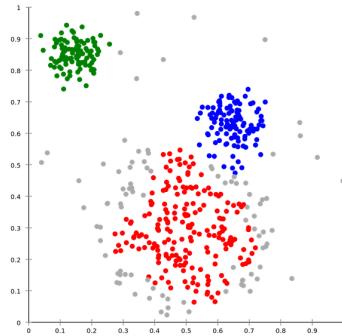


Figure 1.3: Clustering.

tipo supervisado. La diferencia está dada porque no necesitamos conocer el valor de una cierta variable o clase para cada punto, es decir que solo necesitamos los datos en crudo y el algoritmo es capaz de encontrar los clusters automáticamente.

Un problema de clustering se caracteriza, entonces, de la siguiente forma:

- Contamos con un set de datos
- Queremos agrupar este set de datos en clusters/grupos/comunidades

Existe una cierta relación entre los problemas de clustering y de clasificación, por ejemplo dado un problema de clasificación podríamos aplicar clustering primero y luego clasificar a cada punto de acuerdo al cluster al cual pertenece en base a la clase mayoritaria de dicho cluster. Este procedimiento no es muy frecuente pero es conveniente tenerlo en cuenta porque permite entender el funcionamiento de ciertos algoritmos que combinan las propiedades de un problema de clustering y uno de clasificación.

Una aplicación que combina clustering (aprendizaje no supervisado) y clasificación (aprendizaje supervisado) es la que denominamos "Aprendizaje transductivo" en donde usamos los datos para los cuales no conocemos su clase como forma de ayuda a un clasificador tradicional. Consideraremos el siguiente ejemplo:

Tenemos solo dos puntos clasificados, uno clasificado como "blanco" y el otro "negro", sin mas información el punto marcado con el signo de pregunta, cuya clase desconocemos, quedaría clasificado como "blanco" ya que está mucho mas cerca del punto blanco que del punto negro.

Veamos ahora que ocurre si agregamos los puntos cuya clase desconocemos:

Agregando los puntos sin clase vemos que en nuestros datos existen dos clusters y si tenemos que asociar cada cluster con un color entonces el de "arriba" es negro y el de abajo "blanco" ya que el punto negro pertenece al cluster de arriba y el blanco al de abajo. En este caso nuestro punto marcado como "?"

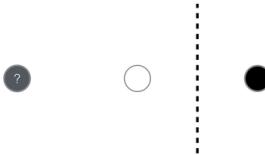


Figure 1.4: Aprendizaje Transductivo.

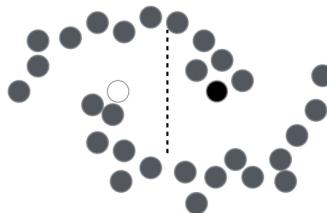


Figure 1.5: Aprendizaje Transductivo.

debería clasificarse como negro ya que está mas cerca del cluster superior que del inferior.

El aprendizaje no-supervisado es una rama muy importante en Data Science ya que solo trabaja con datos sin necesidad de tener un "label" para los mismos, es decir que solo necesitamos los datos en crudo y estos en general son mucho mas fáciles de conseguir que datos ya previamente clasificados. El aprendizaje transductivo es un área muy nueva dentro de Data Science y que sin dudas merece ser explorada.

Recomendaciones

El objetivo de un sistema de recomendaciones es muy simple: recomendarle al usuario cosas que pueden interesarle. El caso mas típico es el de recomendar libros en Amazon, películas en Netflix o productos en un newsletter de un supermercado. También es posible recomendarle a un usuario otros usuarios a seguir en una red social o contenidos que puedan interesarles en un newsreader.

Los sistemas de recomendación son en general particularmente complejos e involucran el esfuerzo conjunto de varios algoritmos y herramientas, tienen realmente muchos detalles a considerar y por eso vamos a dedicarles un extenso tratamiento mas adelante.

Las características de un Sistema de Recomendaciones son:

- Tenemos un conjunto de ítems y un conjunto de usuarios
- Queremos recomendarles a los usuarios ítems que puedan interesarles

Sistemas de Consulta

Un sistema de consulta es simplemente un buscador, un search engine, el contenido que se almacena puede ser de cualquier formato aunque en general se almacenan páginas HTML o texto plano. El objetivo del sistema de consulta es recuperar los textos o páginas o ítems de información mas relevantes para la consulta planteada por el usuario. Todos estamos familiarizados con este tipo de sistemas ya que son los que definen a los buscadores en la web como Google y otros. Las características de un sistema de consultas son:



Figure 1.6: Un search engine conocido.

- Almacenamos información no-estructurada como textos planos, páginas HTML, imágenes, sonido, etc.
- Queremos encontrar los ítems de información mas relevantes para las consultas realizadas

Identificación de Patrones

Este tipo de problemas caracterizan la rama de la computación conocida como "*Data Mining*". El objetivo es descubrir información interesante a partir de un conjunto de datos. El ejemplo más clásico es descubrir asociaciones entre los productos que los clientes compran en un comercio, cosas del estilo "el que compra X frecuentemente también compra Y". A esta tarea se la suele llamar *pattern mining*. En muchos casos los algoritmos de pattern mining encuentran asociaciones inusuales o insospechadas, como por ejemplo que quienes compran pañales en un supermercado también suelen comprar cerveza (un ejemplo muy popular en el mundo de Data Science), tal vez esto se deba a que ambas cosas sirven para solucionar el llanto.

Las características de un sistema de identificación de patrones son:

- Contamos con información que frecuentemente es de tipo transaccional (compras, ventas, visitas a una página, etc)
- Queremos encontrar patrones, asociaciones entre los ítems que se incluyen dentro de cada transacción

Reconocer el tipo de problema que tenemos es sumamente importante ya que nos permite determinar cuales son las armas con las que contamos, es decir los

algoritmos que en general suelen funcionar muy bien para problemas similares al que queremos resolver.

1.1.2 Conseguir los datos necesarios

Conseguir los datos necesarios es un paso fundamental de cualquier proceso de Data Science, en general es la parte que mayor cantidad de tiempo y esfuerzo insume y lamentablemente no suele ser la tarea mas divertida y gratificante. Conseguir los datos no solo pasa por hacerse de la información que necesitamos sino también depurarla y transformarla en el formato que necesitamos para nuestra tarea. Algunos de los puntos clave en la tarea de recolección de los datos son los siguientes:

Obtener los Datos

En esta fase el objetivo es obtener los datos en crudo, en muchos casos esto implica la creación de programas especiales para la captura de los mismos, en otros casos los datos ya están disponibles en una base de datos o en algún formato de archivo. Un detalle importante en esta fase es que en muchos problemas necesitamos un set de entrenamiento y en muchos casos no lo tenemos. Por ejemplo podemos necesitar la calificación de películas para un sistema de recomendaciones o el sentimiento de una buena cantidad de reviews para un sistema de análisis de sentimiento. Lo notable en estos casos es que no es posible automatizar la creación de los *labels* para los datos ya que esto es precisamente lo que pretendemos lograr. Es necesario entonces contar con intervención manual pidiendo a un grupo de humanos que se encargue de crear los labels que necesitamos. Una forma de lograr esto es mediante la contratación del servicio de Amazon conocido como "Mechanical Turk"

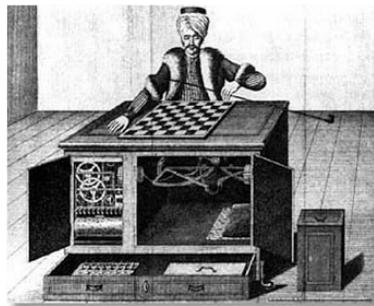


Figure 1.7: Mechanical Turk, un androide ajedrecista del siglo XVIII que da origen al nombre del servicio de Amazon.

En este servicio los usuarios pueden contratar labor humana ofreciendo un cierto precio para realizar tareas repetitivas. Otros usuarios pueden tomar estas

tareas y cobrar el dinero ofrecido luego de terminarlas. Esta forma de *crowdsourcing* es muy popular en la comunidad de Data Science en los casos en los cuales es inevitable contar con intervención humana para crear los sets de entrenamiento. Un factor interesante a analizar es si el tipo de comunidad que suele tomar este tipo de trabajos no introduce algún tipo de *bias*¹ indeseado en los datos.

Para la carga de datos de diversas fuentes es necesario contar con un buen conjunto de bibliotecas que permitan leer datos en diversos formatos: SQL, XML, JSON, CSV, TXT, etc.

Depurar los Datos

La depuración de datos es frecuentemente la parte que insume mayor tiempo en todo proceso de Data Science, suele ser una tarea bastante tediosa y extensa que prácticamente nadie quiere hacer pero es imprescindible ya que de la calidad de los datos va a depender el éxito de nuestra tarea. Depurar los datos implica dejarle a nuestros programas datos que son correctos sintáctica y semánticamente. Los errores a corregir son de todo tipo como por ejemplo: formateo de fechas y formatos numéricos, corrección de caracteres inválidos en strings, conversión de tipos, identificación y solución de valores nulos o faltantes, etc.

Mas allá de la corrección del *valor* de cada dato es necesario también asegurarse que los datos estén *Bien Formados*, es decir que cada "fila" sea una observación y que cada "columna" sea un atributo. Frecuentemente los datos llegan en formatos que no están bien formados y es necesario realizar algunas manipulaciones para transformarlos.

Veamos algunos ejemplos de datos que no están bien formados y la forma de corregirlos.

Ejemplo 1 Nuestro primer ejemplo muestra las notas que los alumnos obtuvieron en ciertas materias:

Alumno	7506	7507
Juan Perez	7	9
Diego Ramirez	6	-
Carlos Marquez	9	8

Table 1.1: Ejemplo 1: Alumnos y Notas.

Este caso es uno de las formas mas comunes de datos mal formados, el problema es que no tenemos cada observación en una fila sino que tenemos varias. Es por esto que el atributo "nota" no aparece como nombre de columna. Para que los datos estén bien formados cada fila tiene que ser una observación y cada columna un atributo. La forma de corregir este set de datos sería la siguiente:

¹Por bias se entiende a una tendencia arbitraria que favorece datos de un cierto tipo

Alumno	Materia	Nota
Juan Perez	7506	7
Juan Perez	7507	9
Diego Ramirez	7506	6
Diego Ramirez	7507	-
Carlos Marquez	7506	9
Carlos Marquez	7507	8

Table 1.2: Ejemplo 1: Alumnos y Notas, datos bien formados.

En esta versión podemos ver que cada observación es una fila y cada atributo es una columna.

Ejemplo 2 En la siguiente tabla mostramos para algunas materias la cantidad de alumnos que obtuvieron notas dentro de un cierto rango

Materia	Aprobado	Bueno	Distinguido	Sobresaliente
7506	5	12	21	9
7507	13	8	4	1

Table 1.3: Ejemplo 2: Alumnos y Notas.

En este caso el problema es que tenemos valores de atributos como nombres de columnas, esto no cumple la definición de datos bien formados. Para corregirlo aplicamos una transformación similar a la de nuestro primer ejemplo:

Materia	Nota	Frecuencia
7506	Aprobado	5
7506	Buenos	12
7506	Distinguido	21
7506	Sobresaliente	9
7507	Aprobado	13
7507	Bueno	8
7507	Distinguido	4
7507	Sobresaliente	1

Table 1.4: Ejemplo 2: Alumnos y Notas datos bien formados.

La versión corregida cumple el principio de que cada observación sea una fila y cada atributo sea una columna.

Existen muchos ejemplos mas de datos mal formados el paper [Wickham] es de lectura recomendable para observar otros ejemplos y la forma de corregirlos.

Detectar y eliminar Anomalías

La detección de anomalías (*outliers*) implica el reconocimiento y corrección o eliminación de datos erróneos, un dato anómalo es aquel que tiene valores imposibles para uno o mas de sus atributos. Decimos que la anomalía es absoluta cuando puede detectarse sin necesidad de contexto, es un error semántico, por ejemplo una persona cuya edad es de 765 años. La anomalía es en cambio relativa cuando el dato es semanticamente posible pero no tiene sentido en el contexto de los demás datos, por ejemplo una distancia de 7000 kilómetros en una base de datos en donde todas las distancias están entre 50 y 80 kilómetros. Probablemente se trate de un dato mal ingresado con dos ceros extra.

La detección de anomalías no siempre es una tarea fácil y son necesarios algoritmos específicos para detectar los puntos anómalos, esto se debe a que en muchos casos los valores de los atributos de un dato pueden parecer razonables pero la combinación de valores es la que resulta anómala. Por ejemplo consideremos una persona que tiene edad 4 años y altura 187cm. Tanto la edad como la altura son valores lógicos y válidos para una persona, lo que no es lógico es la combinación ya que un chico de 4 años no puede medir casi dos metros. Mas adelante veremos un capítulo dedicado a este tema con algoritmos que nos permiten detectar los outliers en un conjunto de datos.

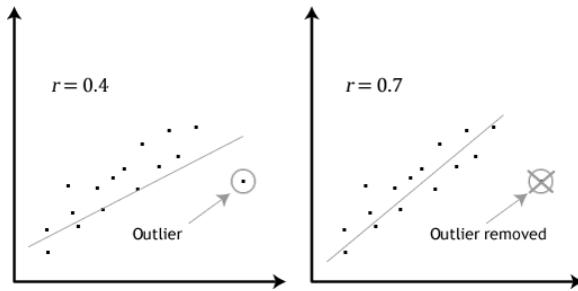


Figure 1.8: Detección de anomalías en un problema de regresión

En la figura 1.8 podemos ver como un dato anómalo puede perjudicar muy seriamente un problema de regresión lineal, el peso de la anomalía cambia completamente el comportamiento de nuestro modelo, ciertos algoritmos son mas sensibles a los outliers que otros, es un dato a tener en cuenta para saber la importancia que debemos darle a la detección de anomalías en nuestro proceso.

Valores Faltantes

En muchos casos, por no decir en una amplia mayoría van a existir datos con valores faltantes para ciertos atributos. Esto puede deberse a que para ciertos datos algunos atributos no tienen sentido o bien a la forma en que los datos fueron recolectados. Por ejemplo en un censo de la población todos los datos

correspondientes al trabajo de una persona no se completan si la persona no tiene trabajo o bien si no quiere informarnos sobre su trabajo.

El problema surge cuando los datos faltantes no son válidos para el algoritmo con el cual vamos a procesar la información. Algunos algoritmos admiten datos incompletos y otros no. En los casos en los que los datos incompletos no son admisibles debemos solucionarlos de alguna forma.

Una opción es eliminar aquellas observaciones para los cuales algún atributo tiene valores faltantes, esto es válido siempre y cuando sean solamente una pequeña minoría dentro de nuestro conjunto de datos. Cuando esto no es posible es necesario de alguna forma completar los atributos faltantes, a esto lo llamamos *imputación*.

El proceso de imputación de valores faltantes puede hacerse de muchas formas, una forma muy simple es completar con el valor promedio o la mediana para atributos numéricos y con el valor más popular para atributos categóricos. Pero también podemos encarar el proceso de imputación como un mini-proceso interno de Data-Science, en cuyo caso completaríamos los atributos numéricos mediante un algoritmo de regresión y los datos categóricos con un algoritmo de clasificación. Lo que hacemos en estos casos es tomar todos los atributos que tiene nuestro dato y en base a estos atributos predecir los atributos faltantes tomando como set de entrenamiento el conjunto de datos para los cuales el atributo tiene algún valor.

Existen varios algoritmos de imputación y algunos lenguajes cuentan con funciones para completar los valores faltantes en los datos automáticamente.

1.1.3 Explorar los datos

En este punto surge lo que llamamos "análisis de datos exploratorio" cuya principal característica es obtener un entendimiento básico de los datos con los cuales se va a trabajar sin tener ningún objetivo en particular. Es decir que simplemente exploramos los datos pero no pretendemos obtener respuestas a partir de los mismos. A continuación explicamos brevemente algunas tareas típicas del análisis exploratorio de datos

Analizar la estructura de los datos

Analizamos la estructura general de los datos para entender con qué información vamos a trabajar.

1. Ver cuantos datos (observaciones) tenemos en total
2. Ver cuantos atributos tiene cada observación
3. Ver el nombre y tipo de dato de cada observación
4. Ver cuantos valores faltantes existen y en qué proporción se presentan para cada atributo

Navegar los datos

Una vez que conocemos los atributos y sus tipos de datos podemos visualizar los datos como si fueran una planilla de cálculo o simplemente como texto. Esta navegación muy básica nos permite entender cuáles son los valores de nuestros atributos, cuáles son los valores más populares, en qué casos hay valores faltantes y detectar algunas posibles anomalías.

Resumen estadístico

Una herramienta fundamental del análisis exploratorio es hacer un resumen estadístico de cada atributo de nuestros datos, algunos datos importantes a conocer son:

1. Valor máximo y mínimo
2. Promedio y media
3. Cantidad de valores faltantes y proporción
4. Quantiles
5. Desviación standard

Visualizar los datos

Esta es una etapa fundamental, la idea es simplemente visualizar gráficamente diferentes aspectos de nuestros datos, podemos realizar todo tipo de ploteos exploratorios, como por ejemplo de un atributo contra otro, o de un atributo solo a lo largo de todo el conjunto de datos. Este análisis es muy útil para determinar varias cosas: en primer lugar podemos ver el grado de correlación entre los atributos, algunos atributos están muy relacionados entre sí mientras que otros son independientes. Podemos también observar posibles anomalías en los datos cuando algunas de nuestras observaciones tengan valores que sobresalen del resto. Finalmente podemos apreciar o inferir el valor predictivo de ciertos atributos. Para ilustrar la importancia del análisis exploratorio vamos a mostrar un caso famoso que se conoce como "el cuarteto de Anscombe" la tabla 1.5 nos muestra los datos en forma tabular.

Tenemos cuatro tablas con dos atributos cada una: X e Y . Si calculamos algunas estadísticas de nuestros datos obtenemos la información de la tabla 1.6.

Como podemos ver todos los valores de X e Y presentan el mismo promedio, suma de valores y desviación standard, es decir que podríamos decir que estadísticamente X_1-Y_1 , X_2-Y_3 , X_3-Y_3 y X_4-Y_4 son iguales. Sin embargo si graficamos estos cuatro plots vemos lo siguiente:

Como podemos ver datos que son estadísticamente idénticos pueden ser completamente diferentes graficamente, por eso es sumamente importante realizar todo tipo de plots para tener un buen conocimiento del conjunto de datos con el que trabajamos. El cerebro humano procesa la información gráfica de forma mucho más eficiente que los números, podemos entender facilmente la "forma"

X1	Y1	X2	Y2	X3	Y3	X4	Y4
10	8.04	10	9.14	10	7.46	8	5.58
8	6.95	8	8.14	8	6.77	8	5.76
13	7.58	13	8.74	13	12.74	8	7.71
9	8.81	9	8.77	9	7.11	8	8.84
11	8.33	11	9.26	11	7.81	8	8.47
14	9.96	14	8.1	14	8.84	8	7.04
6	7.24	6	6.13	6	6.08	8	5.25
4	4.26	4	4.1	4	5.39	19	12.5
12	10.84	12	9.13	12	8.15	8	5.56
7	4.82	6	7.26	7	6.42	8	7.91
5	5.68	5	4.74	5	5.73	8	6.89

Table 1.5: El cuarteto de Anscombe en versión tabular.

VAR	SUM	AVG	STD
X1	99	9	3.32
Y1	82.51	7.5	2.03
X2	99	9	3.32
Y2	82.51	7.5	2.03
X3	99	9	3.32
Y3	82.50	7.5	2.03
X4	99	9	3.32
Y4	82.51	7.5	2.03

Table 1.6: Estadísticas sobre el cuarteto de Anscombe.

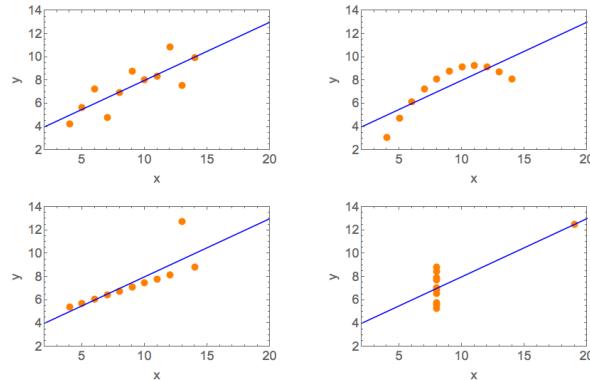


Figure 1.9: Visualización del cuarteto de Anscombe

de los datos del cuarteto de Anscombe al ver los plots pero esta forma no es tan

fácil de visualizar si solamente vemos la tabla con los datos en forma numérica. Estos plots no tienen que ser ni lindos ni tampoco detallados son simplemente herramientas para rápidamente dar un vistazo gráfico a los datos.

Uno de los plots mas usados para el análisis exploratorio es el que nos muestra la correlación que existe entre las variables de nuestro set de datos. Veamos un ejemplo de este plot para el set de datos "Iris".

Iris es un set de datos sobre flores en el que hay cuatro variables y una clase. Las variables son el ancho y alto de los pétalos y el ancho y alto de los sépalos. El objetivo de este set de datos es ver si con esta información es posible determinar a que especie pertenece la flor, las especies son en total 4. Para hacer el plot de correlación usamos únicamente los atributos, es decir que quitamos de nuestro set de datos el atributo que indica la especie a la cual pertenece la flor.

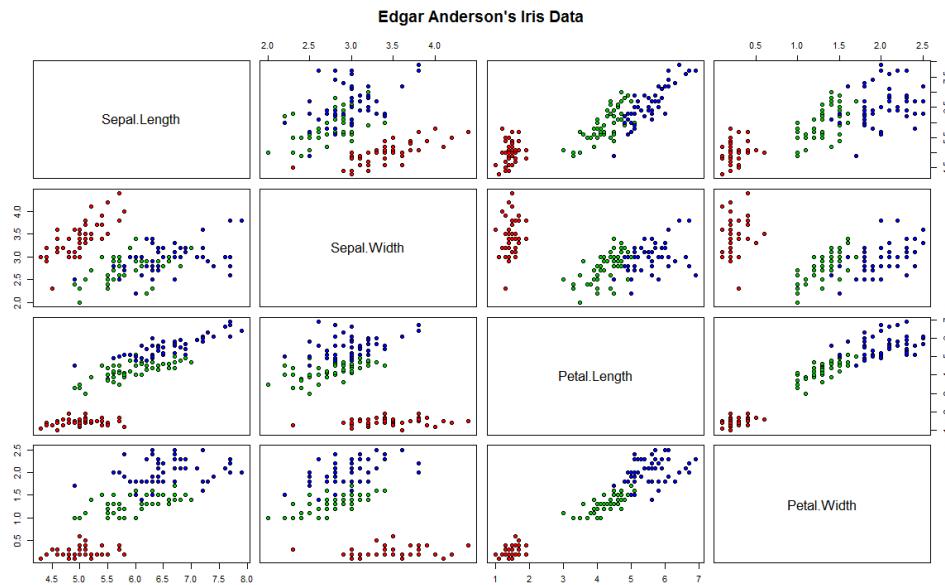


Figure 1.10: Plot de correlación Para Iris

Entender este plot es muy simple, tenemos cuatro filas y cuatro columnas correspondientes a los cuatro atributos de nuestro set de datos, cada "celda" M_{ij} de nuestra matriz representa el plot del atributo i contra el atributo j , la diagonal tiene simplemente el nombre del atributo ya que no tiene sentido plotear un atributo contra si mismo.

Para "Iris" podemos ver que existe una correlación lineal bastante fuerte entre el atributo "3" y el "4" es decir el largo y alto de los pétalos de las flores. También parece existir una correlación entre el largo de los sépalos (atributo 1) y el largo de los pétalos (atributo 3). Esto nos permitiría suponer que podemos usar tres o incluso dos de los atributos que tenemos para predecir la especie

a la cual pertenece la flor es decir que nuestro set de datos que originalmente está en cuatro dimensiones podría funcionar en tres o dos dimensiones. Estas observaciones sobre la dimensionalidad de los datos serán muy importantes mas adelante.

1.1.4 Aplicar a los datos el algoritmo necesario

Una vez que los datos han sido depurados estamos en condiciones de aplicar el o los algoritmos necesarios para responder las preguntas que tenemos. Esta es la fase mas interesante de todo el proceso, donde realmente ocurre la magia, no vamos a hablar en detalle sobre los algoritmos que podemos aplicar ya que todo el resto del curso trata sobre este tema.

1.1.5 Comunicar los resultados

La última etapa del proceso es comunicar los resultados logrados, las respuestas a las preguntas que nos hicimos en un principio. En general esto implica la creación de reportes o visualizaciones que nos permitan transmitir las conclusiones a las que hemos llegado a partir de los datos. Como hemos mencionado previamente las personas entienden los gráficos de una forma mucho mas eficiente que una tabla de números o una narrativa, es por esto que la comunicación de resultados esta dominada por visualizaciones. Vamos a dedicar un extenso capítulo a la visualización de datos mas adelante.

1.2 Formatos de Datos

En esta sección vamos a comentar y analizar brevemente formatos de datos que son populares en Data Science, estos formatos extienden en cierta forma las estructuras de datos tradicionales como vectores, matrices y listas.

1.2.1 Data Frames

El *Data Frame* es la estructura de datos más común e importante en un proceso de Data Science, es equivalente a una tabla en donde tenemos cada dato/observación como una fila y cada atributo como una columna. Cada columna tiene un tipo de dato, no es válido tener en una misma columna datos de diferente tipo.

Hay dos forma de ver a un data-frame: como un conjunto de filas o como un conjunto de columnas. Si pensamos en un data-frame como un conjunto de columnas entonces el data-frame es una lista de vectores en donde cada vector representa a un atributo (columna). Desde el punto de vista de la semántica de las estructuras de datos vamos a considerar que todos los datos en un vector deben ser del mismo tipo mientras que los datos dentro de una lista pueden ser de diferente tipo. Si consideramos a un data-frame como un conjunto de filas entonces el data-frame es un vector de diccionarios en donde cada diccionario representa los datos de cada una de nuestras observaciones o datos.

Cada columna de un dataset puede ser de los siguientes tipos de datos:

survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	3	male	22.0	1	0	7.25	S	Third	man	True		Southampton	no	False
1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
1	3	female	26.0	0	0	7.925	S	Third	woman	False		Southampton	yes	True
1	1	female	35.0	1	0	53.1	S	First	woman	False	C	Southampton	yes	False
0	3	male	35.0	0	0	8.05	S	Third	man	True		Southampton	no	True
0	3	male		0	0	8.4583	Q	Third	man	True		Queenstown	no	True
0	1	male	54.0	0	0	51.8625	S	First	man	True	E	Southampton	no	True
0	3	male	2.0	3	1	21.075	S	Third	child	False		Southampton	no	False
1	3	female	27.0	0	2	11.1333	S	Third	woman	False		Southampton	yes	False
1	2	female	14.0	1	0	30.0708	C	Second	child	False		Cherbourg	yes	False
1	3	female	4.0	1	1	16.7	S	Third	child	False	G	Southampton	yes	False
1	1	female	58.0	0	0	26.55	S	First	woman	False	C	Southampton	yes	True
0	3	male	20.0	0	0	8.05	S	Third	man	True		Southampton	no	True
0	3	male	39.0	1	5	31.275	S	Third	man	True		Southampton	no	False
0	3	female	14.0	0	0	7.8542	S	Third	child	False		Southampton	no	True
1	2	female	55.0	0	0	16.0	S	Second	woman	False		Southampton	yes	True
0	3	male	2.0	4	1	29.125	Q	Third	child	False		Queenstown	no	False
1	2	male		0	0	13.0	S	Second	man	True		Southampton	yes	True
0	3	female	31.0	1	0	18.0	S	Third	woman	False		Southampton	no	False
1	3	female		0	0	7.225	C	Third	woman	False		Cherbourg	yes	True
0	2	male	35.0	0	0	26.0	S	Second	man	True		Southampton	no	True
1	2	male	34.0	0	0	13.0	S	Second	man	True	D	Southampton	yes	True
1	3	female	15.0	0	0	8.0292	Q	Third	child	False		Queenstown	yes	True

Example:Titanic Dataset

Figure 1.11: Titanic Dataset

- Caracteres: Strings o texto que pueden tomar cualquier valor, por ejemplo el nombre de una persona.
- Numericos: Enteros o flotantes (subtipos) por ejemplo la edad de una persona o el precio de un artículo
- Factores: Datos categóricos para los cuales existe una lista de valores posibles, por ejemplo Si/No
- Fechas: Datos en formato de fecha o timestamp, pueden incluir Año, Mes, Dia, Hora, Minutos y Segundos.

En el set de datos "Titanic" tenemos columnas de diferentes tipos. "Class" es un factor que puede ser "First", "Second", "Third" o "Crew" indicando a que clase pertenecía el pasajero. "Sex" es un factor que indica el género del pasajero y puede ser "Male" o "Female". "Age" es una columna numérica que indica la edad del pasajero. "Survived" es un factor que indica si el pasajero sobrevivió a la tragedia.

Desde el punto de vista de las estructuras de datos el único tipo especial son los factores. Para representar un factor necesitamos un vector de valores

posibles y luego podemos codificar el valor que cada dato toma para dicho factor mediante un índice al vector de valores posibles. De esta forma si el atributo "sex" de una persona puede ser "Male" o "Female" podemos representar con 1 el valor "Male" y con 2 el valor "Female" (o con 0 y 1 según convención).

Los tipos de datos mas comunes en un data frame son entonces:

1. Numéricos y continuos
2. Numéricos y discretos
3. Texto
4. Fechas
5. Categóricos o Factores (n valores posibles) no-ordinales (el orden no importa ejemplo: red, blue, green)
6. Categóricos o Factores (n valores posibles) ordinales (el orden importa ejemplo: low, medium, high)

Codificación de Atributos Categóricos

Consideremos el caso de atributos categóricos es decir en el cual tenemos una lista de valores posibles. Algunos ejemplos son atributos de tipo Si/No, provincia, sexo, color de un objeto, la clase a la cual pertenecía un pasajero del Titanic, etc. En todos los casos cada observación toma para el atributo en cuestión un valor que pertenece a una lista de valores posibles.

En muchos casos a un algoritmo le resulta confuso entender los datos de tipo categóricos por lo que es conveniente transformarlos en un formato equivalente pero mas simple de procesar para el algoritmo. Vamos a ver tres posibles ejemplos de codificación para los siguientes datos:

Pasajero	Clase
John Smith	First Class
Susan Walters	Third Class
Megan Smith	First Class
Willie Wilburg	Second Class

Table 1.7: Codificación de atributos categóricos.

Nuestro atributo categórico puede tomar 3 valores: "First Class", "Second Class" y "Third Class".

Una primera aproximación es simplemente reemplazar el atributo por un número que indica a cual de los n valores corresponde el mismo. A esto lo llamamos **codificación ordinal**

La codificación ordinal no suele tener muy buenos resultados pero es el método que menor cantidad de columnas agrega a nuestros datos, en los casos en los cuales la cantidad de valores posibles para el atributo es enorme puede ser la única representación posible.

Pasajero	Clase
John Smith	1
Susan Walters	3
Megan Smith	1
Willie Wilburg	2

Table 1.8: Codificación Ordinal.

La siguiente codificación se denomina **”one hot encoding”** y consiste en dividir el atributo en tantas columnas como valores posibles puede tener y usar cada columna como un dato binario indicando si el atributo toma o no dicho valor.

Pasajero	First Class	Second Class	Third Class
John Smith	1	0	0
Susan Walters	0	0	1
Megan Smith	1	0	0
Willie Wilburg	0	1	0

Table 1.9: One Hot Encoding.

One-hot encoding es la codificación mas popular, se usa de forma casi ubicua en la representación de datos para algoritmos de Data Science.

Una tercera variante consiste en convertir los datos a un número como hicimos en la codificación ordinal y luego representar dicho número como un binario de $\log(k)$ bits siendo k la cantidad de valores que puede tomar el atributo. Luego representamos cada bit como una columna binaria.

Pasajero	bit1	bit0
John Smith	0	1
Susan Walters	1	1
Megan Smith	0	1
Willie Wilburg	1	0

Table 1.10: Codificación binaria.

La **codificación binaria** es un compromiso entre one-hot encoding y la codificación ordinal. Funciona mejor que la codificación ordinal pero requiere menos columnas que one-hot encoding. Notar que usamos 01,10 y 11 para las clases pero podríamos haber usado 00,01 y 10 respectivamente.

1.2.2 Texto

Los textos en formato plano son uno de los formatos mas simples y habituales en un proceso de Data-Science, a partir de un conjunto de textos podemos ex-

traer una gran cantidad de información, por ejemplo podemos realizar consultas, buscar frases o palabras, hacer un resumen automático de textos, analizar las palabras (keywords) o frases mas relevantes, responder preguntas en lenguaje natural, etc. Todas los procesos que involucran texto necesitan que de alguna forma transformemos los textos en un conjunto de datos que nuestros algoritmos puedan procesar, en esta sección introducimos el modelo mas simple para modelar textos denominado "bag of words" (BOW). Mas adelante veremos modelos mucho mas sofisticados.

Bag of Words (BOW)

En el modelo BOW cada texto se representa con un vector de "n" dimensiones siendo "n" la cantidad total de términos (palabras) en la colección. De esta forma si hay 300.000 palabras en total cada texto está representado por un vector de 300.000 dimensiones. El vector contiene simplemente la cantidad de veces que cada palabra aparece en el texto.

Consideremos el siguiente ejemplo:

Once there was a king
the king was good

Suponiendo que no hay mas texto que lo que vemos tenemos un vocabulario (léxico) de 7 palabras:

1. once
2. there
3. was
4. a
5. king
6. the
7. good

Es decir que podemos representar a nuestros textos con vectores de 7 elementos. Si consideramos a cada línea de nuestro texto como un documento independiente entonces tendríamos dos vectores:

$(1, 1, 1, 1, 1, 0, 0)$
 $(0, 0, 1, 0, 1, 1, 1)$

El primer elemento de cada vector corresponde a la primera palabra y así sucesivamente. En este ejemplo muy simple tenemos muy pocas palabras y muy pocos documentos en general para un conjunto de documentos grandes los

vectores pueden llegar a tener cientos de miles de dimensiones y son vectores muy dispersos ya que la mayoría de sus elementos son ceros. Esto se debe a que es mucho mas probable que un documento no tenga una palabra cualquiera a que la tenga.

1.2.3 Datos Matriciales

Una matriz es una simplificación de un data-frame, la matriz es un conjunto de valores numéricos formateados en "m" filas y "n" columnas. Es interesante destacar que un gran número de problemas de Data-Science se presentan en forma de matrices. Es de destacar que excepto los strings todos los tipos de datos que hemos visto se pueden asociar a un valor numérico, los factores se pueden codificar con one-hot encoding o alguna otra de las codificaciones vistas, las fechas pueden convertirse en un offset en segundos a partir de un cierto EPOCH y los valores numéricos no hace falta convertirlos. Es por esto que en la mayoría de los casos podemos convertir el set de datos en una matriz numérica con la cual trabajar.

Matrices Dispersas

En muchos casos las matrices con las cuales trabajamos son dispersas es decir que tienen una gran mayoría de ceros y muy pocos elementos distintos de cero. En estos casos es necesario contar con formatos especiales que nos permitan almacenar la matriz de forma eficiente. En modo texto un formato de archivo que es eficiente para almacenar matrices dispersas es el formato de LIBSVM en el cual cada línea del texto representa a una fila de la matriz, el formato de LIBSVM establece que el primer elemento de cada línea es el "label" correspondiente a la fila y luego se indican los elementos distintos de cero con la notación *col : value* es decir número de columna y valor.

Por ejemplo si tenemos el siguiente archivo LIBSVM:

```
label1 2:4 5:3
label2 1:3
label3 2:2
```

La matriz representada es la siguiente:

$$\begin{pmatrix} 0 & 4 & 0 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \end{pmatrix}$$

El formato es sencillo de entender. En memoria hay varias formas de trabajar con matrices dispersas, un mecanismo que suele funcionar bien y es muy usado es el llamado "compressed row storage" (CRS).

CRS (Compressed Row Storage)

Usando CRS una matriz dispersa se almacena mediante tres vectores. El primer vector (V_1) almacena los elementos distintos de cero de la matriz. El segundo vector (V_2) almacena los números de columna en los que aparecen los elementos distintos de cero de la matriz. El tercer vector (V_3) indica el índice al primer vector para los elementos que son el inicio de una nueva fila en la matriz.

Para la matriz de nuestro ejemplo los vectores serían los siguientes:

$$\begin{aligned} V_1 &= (4, 3, 3, 2) \\ V_2 &= (2, 6, 1, 2) \\ V_3 &= (1, 3, 4) \end{aligned}$$

Notar que en todos los casos usamos como convención numerar filas y columnas a partir de 1, indistintamente puede numerarse a partir de cero.

Los vectores V_1 y V_2 tienen igual cantidad de elementos: tantos como elementos distintos de cero haya en la matriz original. El vector V_3 tiene tantos elementos como filas tenga la matriz. Usando CRS es posible recuperar cualquier fila de la matriz sin necesidad de recorrerla completamente. Por ejemplo si queremos la fila 2 de la matriz sabemos que el segundo elemento de V_3 nos apunta al primer elemento distinto de cero de la fila 2, es decir que el primer elemento distinto de cero es el tercer elemento de V_1 y V_2 es decir un 3 en la columna 1. Sabemos que la fila 2 solo tiene un elemento distinto de cero ya que el siguiente elemento de V_3 es el índice 4 y corresponde a la fila 3, por lo tanto la fila 2 de la matriz es:

$$(3, 0, 0, 0, 0, 0)$$

1.2.4 Imágenes

Las imágenes también suelen ser un set de datos habitual en problemas de Data Science, una imagen se representa en general mediante un tensor de $X * Y * 3$ elementos. Siendo X e Y el largo y alto de la imagen en pixeles. El "3" se debe a que por cada pixel almacenamos tres valores correspondientes a los canales RGB, de esta forma cada pixel de la imagen queda representado por el valor de intensidad del mismo para el canal rojo, verde y azul que en general se representa mediante un número de 8 bits lo cual nos da una profundidad de color de 24 bits.

1.3 Niveles de Almacenamiento

En todo proceso de análisis y procesamiento de datos podemos distinguir tres niveles de almacenamiento, tres diferentes lugares en donde pueden estar alojados los datos que tenemos que procesar.

1. En Memoria

2. En Disco
3. En un Cluster

1.3.1 Datos en Memoria

La memoria es el medio mas eficiente en cuanto a velocidad para almacenar los datos pero también es el mas costoso, aún así hoy en día muchos, tal vez la mayoría de los sets de datos, pueden ser alojados en memoria para ser procesados de forma eficiente.

Cuando los datos están alojados en memoria podemos organizarlos usando diferentes estructuras de forma tal de poder acceder a un dato buscado sin necesidad de recorrer todos los datos. Incluso en memoria recorrer todos los datos puede ser un proceso costoso. Algunas de las estructuras de datos mas populares para almacenar datos en memoria son:

1. Tablas de hash (diccionarios)
2. Arboles binarios y variantes
3. Tries y variantes
4. Listas
5. Indices Invertidos
6. Skip-Lists

En general es conveniente tener un manejo adecuado de estas estructuras de datos para poder operar eficientemente con datos a nivel de memoria. En concreto los diccionarios y las tablas de hash son estructuras que vamos a usar muy frecuentemente. Algunos de estos temas los vamos a desarrollar en los siguientes capítulos.

1.3.2 Datos en Disco

Los datos se almacenan en disco como forma de soporte permanente ya que la memoria es volátil, la forma de almacenar los datos en disco puede ser mediante archivos o bien una base de datos. Existen diversos tipos de archivos y bases de datos para persistir datos en disco, las bases relationales son las mas populares y durante muchos años fueron la herramienta ubicua para almacenar y procesar datos. Luego aparecieron las bases No-SQL que son bases de datos especializadas en cierto tipo de información como por ejemplo documentos de texto o grafos. Mas adelante dedicaremos algo de tiempo a ver este tipo de bases de datos.

En una gran cantidad de problemas las bases de datos no son necesarias y se usan simples archivos para almacenar los datos, estos archivos pueden ser de tipo .CSV, .TXT, JSON u otros formatos. Los formatos basados en XML

que fueron muy populares, hoy en día han caído en desuso ya que agregan la complejidad del proceso de XML a la complejidad propia de los datos.

En general no es una buena idea operar sobre datos almacenados en disco ya que las velocidades de acceso y transferencia de los mismos son bastante lentas. El modelo más usado consiste en levantar los datos a procesar en memoria y realizar los procesos necesarios repitiendo este proceso tantas veces como sea necesario. En los contados casos en los cuales es necesario procesar los datos sobre el disco mismo es conveniente recordar que la forma más eficiente de procesar datos en disco es accediendo a los mismos de forma secuencial. Dos o incluso tres lecturas secuenciales suelen ser más eficientes que leer de forma aleatoria es decir accediendo a los datos mediante referencias a los mismos. Esto es entendible debido a que en una lectura secuencial no hace falta realizar "seeks" que es una operación costosa en un disco. En los dispositivos SSD que no tienen necesidad de hacer "seek" leer secuencialmente o de forma aleatoria es más o menos equivalente.

1.3.3 Datos en un Cluster

Cuando los datos son tan masivos para exceder la capacidad de almacenamiento de uno o varios discos no queda más remedio que almacenarlos en un cluster, es decir distribuidos en varios equipos. Los volúmenes de información que se manejan actualmente hacen que esta forma de almacenamiento y procesamiento de datos sea cada vez más popular y necesaria por lo que dedicaremos el próximo capítulo a los pormenores y detalles del almacenamiento y procesamiento distribuido.

1.4 La Ley de Zipf y las Leyes de Potencias

La ley de Zipf describe datos en los cuales hay muy pocos eventos de mucha magnitud (impacto) y muchos eventos de muy baja magnitud. El ejemplo más tradicional es la frecuencia de las palabras de un determinado idioma en un texto o conjunto de textos. Algunas palabras, que son muy pocas, como "the" aparecen una enorme cantidad de veces mientras que hay una enorme cantidad de palabras con frecuencia muy baja. Otro ejemplo es la cantidad de visitas a sitios web. Algunos pocos sitios web tienen muchísimos visitantes mientras que existen millones de sitios con muy pocas visitas.

La figura 1.12 nos muestra el gráfico de visitas para usuarios de AOL en un cierto período de tiempo. El gráfico tiene una forma de "L" casi perfecta, hay muchos sitios con muy pocas visitas (users) y muy pocos sitios con muchísima cantidad de visitas.

Este gráfico es muy difícil de interpretar, la verdadera aparición de la ley de Zipf y las leyes de potencias se evidencia si graficamos usando en los ejes escalas logarítmicas.

En la figura 1.13 vemos el mismo plot en escala logarítmica como podemos ver el gráfico es ahora una recta, con algo de ruido en la parte final. Esta es

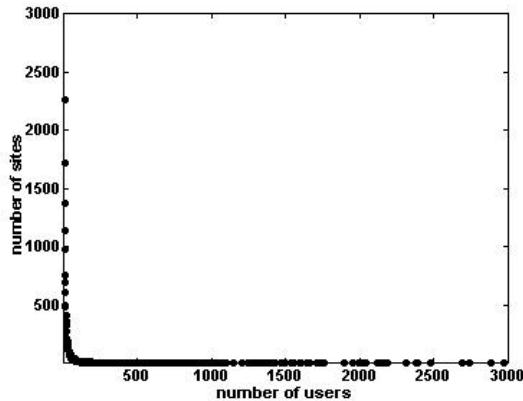


Figure 1.12: Cantidad de visitas a sitios web

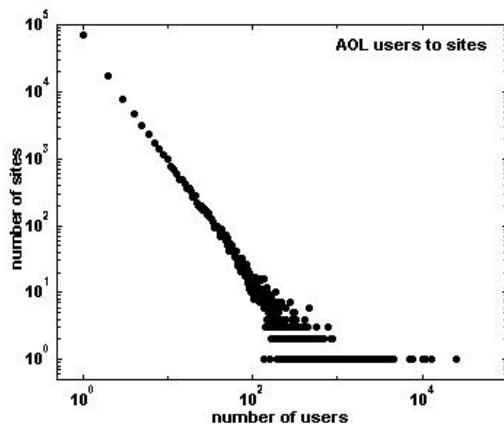


Figure 1.13: Cantidad de visitas a sitios web. Log-Log plot.

la marca característica de lo que llamamos una *ley de potencias* (power-law) y que, como viremos, es sinónimo de una distribución que obedece la ley de Zipf.

Cada vez que un plot en escala logarítmica presenta una recta existe la confusión de si estamos viendo una ley de potencias, la ley de zipf o la distribución de Pareto. En realidad son tres formas diferentes de interpretar el mismo tipo de distribución, a continuación explicamos cuáles son sus definiciones y equivalencias para dejar en claro estos conceptos que son muy importantes.

1.4.1 Pareto, Zipf y Power-Laws

La ley de Zipf fue enunciada por George Kingsley Zipf un linguista de Harvard que estaba interesado en determinar la frecuencia de la i -ésima palabra mas frecuente en un cierto idioma. Se define, entonces, en base al ranking de cada uno de los eventos ordenados por frecuencia y dice que la frecuencia de un evento es inversamente proporcional a su ranking. Esto quiere decir que si en un texto tenemos la palabra "the" que ocurre 4000 veces podemos esperar que la palabra siguiente en frecuencia ocurra 2000 veces la siguiente $4000/3$ veces etc. En definitiva si llamamos f a la frecuencia de un evento tenemos que la frecuencia puede calcularse como:

$$f \approx r^{-b} \quad (1.1)$$

En donde el exponente b es un valor muy cercano a 1. Esto quiere decir que si ploteamos en el eje X el ranking y en el eje Y la frecuencia y usamos una escala logarítmica obtenemos una recta cuya pendiente es muy similar a -1.

Por lo tanto la *ley de Zipf* y las distribuciones Zipfianas se caracterizan por estudiar frecuencia en función de ranking y por presentar una recta cuya pendiente es -1 en un gráfico de escala logarítmica.

Pareto estaba interesado en la distribución de los ingresos entre la población. En lugar de preguntarse cuál es el i -ésimo ingreso mas alto le interesaba saber cuántas personas tenían un ingreso mayor a un cierto valor x . La *ley de Pareto* está definida entonces en función de una función de acumulación de la distribución (CDF) y dice que la cantidad de eventos cuya frecuencia es mayor a x es una potencia inversa de x .

$$P[X > x] \approx x^{-k} \quad (1.2)$$

Esta ley nos indica que hay muy pocos billonarios y que la enorme mayoría de la población tiene ingresos pequeños.

Las *leyes de potencias* (power laws) son la versión mas simple de estos tres puntos de vista y nos dicen simplemente cuál es la frecuencia de un evento cuyo valor es x . Por ejemplo cuántas personas tienen un cierto ingreso x o cuántos websites tuvieron una cierta cantidad de visitas. Es simplemente la distribución de probabilidades (PDF) asociada a la ley de Pareto.

$$P[X = x] \approx x^{-k+1} = x^{-\alpha} \quad (1.3)$$

Es decir que $\alpha = k + 1$ en donde k es el exponente de la ley de Pareto. Esto establece que la ley de Pareto es una ley de potencias. A continuación vamos a mostrar que también hay una relación directa entre la ley de Zipf y la ley de Pareto. En una ley de potencias tenemos que $y = Cx^{-\alpha}$ lo cual implica que $\log(y) = \log(C) - \alpha \log(x)$. Esto nos muestra que una ley de potencias con exponente α es una recta cuya pendiente es $-\alpha$ en escala logarítmica. Podríamos tentarnos y decir que la recta que mejor ajusta al plot de los datos en escala logarítmica es una buena forma de obtener el exponente α asociado a la ley de potencias pero esto no sería una buena idea.

El problema es que la cola final de la distribución (ver 1.13) es ruidosa, esto es porque a medida que aumentamos la frecuencia cada vez hay menos eventos que tienen dicha frecuencia. En el caso de nuestro ejemplo hay muy pocos websites que tienen una gran cantidad de visitas.

La forma de corregir esto es discretizar los datos en intervalos cada vez mas grandes, por ejemplo el segundo intervalo puede ser el doble del primero, el tercero 4 veces el primero, etc. Luego dividimos la cantidad de puntos que caen dentro del intervalo por el ancho del mismo para normalizar. Con este procedimiento la cantidad de puntos en cada intervalo se hace pareja. En nuestro ejemplo a medida que pedimos mayor cantidad de visitas (users) agrupamos a mayor cantidad de sitios. Si es necesario un ejemplo numérico podemos imaginar que el primer intervalo va de 0 a 1 visita el segundo de 2 a 4 visitas el tercero de 4 a 8 visitas, etc. Los intervalos son cada vez mas grandes por lo que la cantidad de sitios va a ser mas pareja en cada uno.

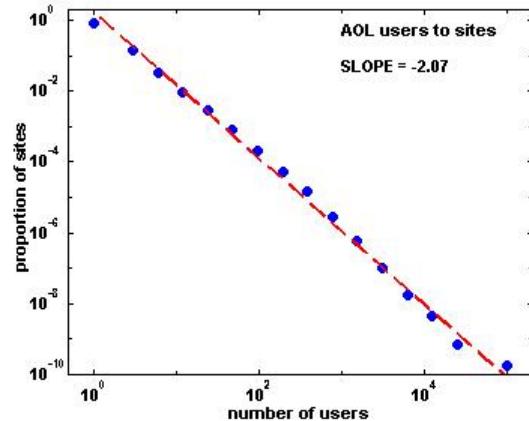


Figure 1.14: Cantidad de sitios por cantidad de visitas discretizada logarítmicamente en escala log-log normalizada

La figura 1.14 nos muestra el resultado de hacer esto con los datos de las visitas a los websites. Podemos ver que hay mucho menos ruido. Si ajustamos ahora una recta al gráfico obtenemos una pendiente de -2.07.

Otra forma de calcular el exponente de una ley de potencias es mediante la siguiente fórmula (derivada de la formulación de maximum likelihood para $p(x)$)

$$\hat{\alpha} = 1 + n \left(\sum_{i=1}^n \ln \frac{x_i}{x_{min}} \right)^{-1} \quad (1.4)$$

Una alternativa al procedimiento que hemos descripto es usar la distribución de Pareto $P[X > x] = x^{-k}$ el ruido de la cola se suaviza de forma natural en la

acumulación de la distribución.

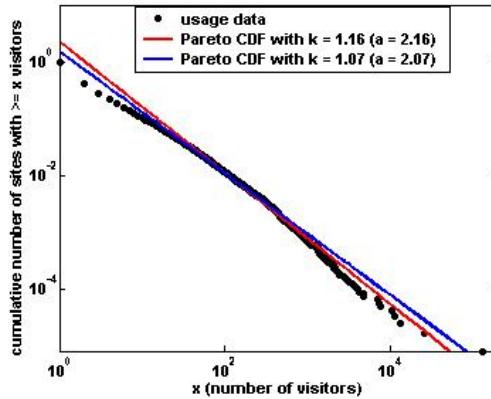


Figure 1.15: Distribución acumulada

En la figura 1.15 vemos el gráfico en escala logarítmica de la cantidad de visitas en el eje x y el acumulado de número de sitios con al menos esa cantidad de visitas en el eje y (Pareto). Si ajustamos una recta a este gráfico obtenemos como pendiente $\alpha = -2.16$ que es un valor muy similar al que habíamos obtenido discretizando la ley de potencias.

Grafiquemos ahora la cantidad de visitas en función del ranking del sitio (de 1 a N).

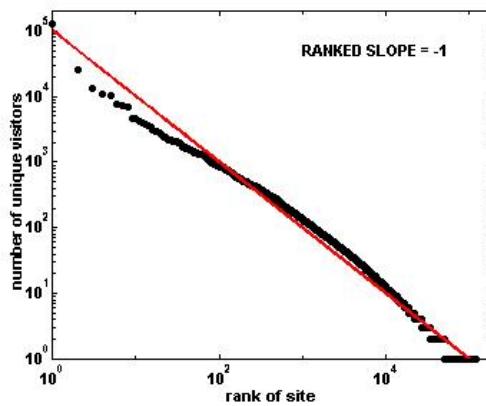


Figure 1.16: Ley de Zipf

Podemos observar en la figura 1.16 que en escala logarítmica el plot tiene pendiente similar a -1 lo que nos indica que este set de datos cumple la ley de

Zipf.

Podríamos pensar que tenemos dos leyes de potencias diferentes: una para la distribución de las frecuencias y otra para la ley de Zipf (ranking). La clave es formular el ranking de la forma adecuada para ver su relación con Pareto. La frase "El r-ésimo sitio mas visitado tiene n visitas" es equivalente a decir "r sitios tienen n o mas visitas" que es exactamente la definición de la distribución de Pareto pero con los ejes x e y intercambiados. Para Zipf tenemos a "r" (ranking) en el eje x y n (visitas) en el eje y mientras que para Pareto tenemos n (visitas) en el eje x y r en el eje y. Por lo tanto:

$$n \approx r^{-b} (\text{Zipf})$$

Y entonces el exponente de Pareto es $1/b$ y vale:

$$r \approx n^{-1/b} (\text{Pareto})$$

En esta última fórmula n es cantidad de visitas y r es cantidad de sitios que tienen n o mas visitas.

Como hemos visto que la ley de potencias es una derivación directa de Pareto su exponente será $1 + 1/b$ lo cuál nos dice que Pareto, Zipf y la ley de Potencias son puntos de vista diferentes para el mismo fenómeno y la relación que existe para calcular uno en función de cualquiera de los otros.

1.4.2 Leyes de Potencias

El fenómeno que podemos interpretar como una ley de potencias, Zipf o Pareto se da en muchísimas situaciones en el mundo real. Algunas distribuciones de este tipo son:

1. Frecuencia de las palabras en la mayoría de los lenguajes.
2. Cantidad de citas de papers científicos.
3. Cantidad de visitas a sitios web.
4. Cantidad de ventas de libros.
5. Cantidad de llamadas por línea telefónica.
6. Magnitud de terremotos.
7. Diámetro de cráteres lunares.
8. Intensidad de las tormentas solares.
9. Cantidad de amigos en una red social.
10. Frecuencias de nombres de personas.
11. Población de ciudades.
12. Ingreso neto de la población.

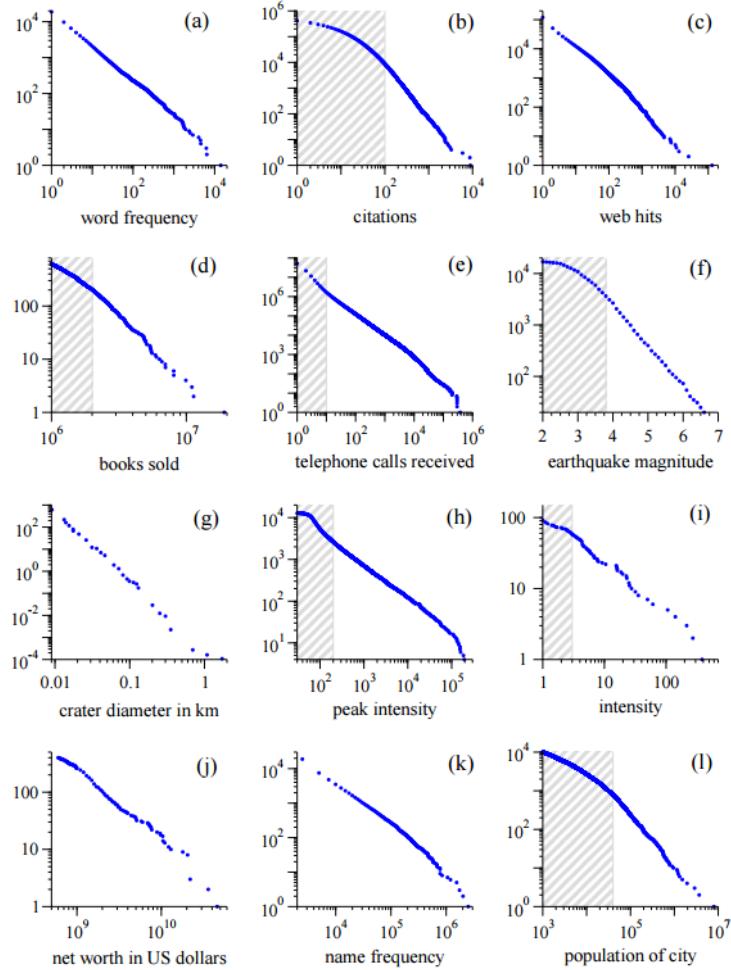


Figure 1.17: Plots de ranking/frecuencia para diferentes fenómenos. En escala logarítmica podemos ver que la pendiente de todos es muy similar a -1 lo cual evidencia que cumplen la ley de Zipf.

1.4.3 Propiedades de las Leyes de Potencias

Un Power Law se caracteriza por una distribución de la forma:

$$p(k_i = x) = x^{-\alpha} \quad (1.5)$$

Necesidad de un valor mínimo

La primera propiedad importante es que es necesario fijar un valor mínimo para x ya que las leyes de potencias divergen si x es un valor cercano a 0. Por ejemplo para las frecuencias de palabras funciona usar 1 como frecuencia mínima. Para la magnitud de los terremotos podríamos usar 3.8 en la escala de Richter, etc.

Diferencias con una exponencial

En segundo lugar veamos en que se distingue una ley de potencias de una función de la familia exponencial:

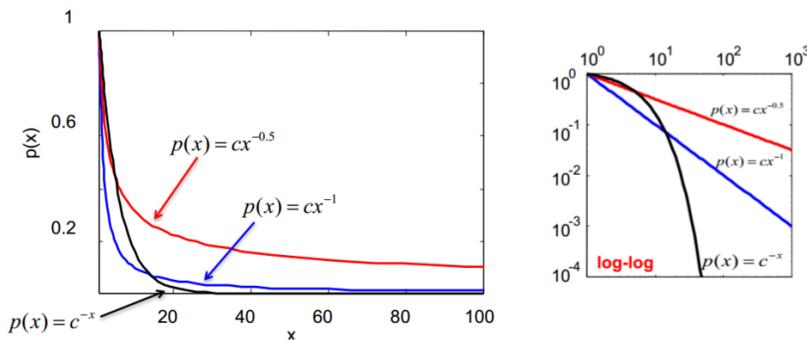


Figure 1.18: Power Law vs Exponential

La diferencia con una distribución exponencial es que el exponente es una constante. Al graficar la distribución del grado es difícil distinguirla de una exponencial, la diferencia matemática es que existe un valor x a partir del cual la exponencial siempre es menor que una ley de potencias, esto es muy fácil de demostrar pero es muy difícil apreciarlo en un gráfico. La mejor forma de detectar una ley de potencias es hacer el gráfico con escala logarítmica, en cuyo caso la ley de potencias aparece como una recta mientras que una exponencial es una curva.

Dependencia de los eventos

Podemos afirmar que los eventos que responden a una ley de potencias dependen entre sí. Si los eventos fueran independientes entonces su distribución acumulada sería normal de acuerdo al Teorema Central del límite. En el caso de las palabras la cantidad de veces que aparece una determinada palabra depende de las demás palabras. En otros casos esto es mas difícil de ver, en el caso de las llamadas telefónicas podemos afirmar que la cantidad de veces que se llama a un cierto número depende de los otros números, tal vez la forma correcta de explicar esto sea pensar que el que llama a muchos números tiene menos tiempo de llamar a otros.

Invariancia a la escala

Las leyes de potencias son invariantes a la escala. Si tenemos $y = x^{-\alpha}$ multiplicar a x por una constante c nos da: $cx^{-\alpha} = c^{-\alpha}x^{-\alpha} \approx x^{-\alpha}$

Promedio y varianza

Las leyes de potencias solo tienen promedio si el exponente α es mayor a 2 y tienen varianza infinita si el exponente es menor a 3. La mayoría de las leyes de potencias tienen α entre 2 y 3 por lo que tendrán promedio pero varianza infinita. La varianza infinita da origen al fenómeno del Cisne Negro (Black Swan).

Fenómeno del Cisne Negro

El fenómeno del Cisne Negro ocurre cuando un evento es inesperado pero al mismo tiempo lógico. Si vemos un cisne blanco, luego otro cisne blanco, luego otro, y así durante miles de eventos cuando aparezca un cisne negro nos llevaremos una tremenda sorpresa. Sin embargo esto era lógico porque en la naturaleza hay tanto cisnes negros como blancos.

1.4.4 Origen de las leyes de potencias

El motivo por el cual ciertos eventos responden a una ley de potencias no es claro, hay muchas teorías y estudios al respecto pero ninguno es conclusivo. En su tratado original Zipf justificó su ley basándose en el *principio del mínimo esfuerzo*. Cuando dos personas se comunican usando un cierto lenguaje intentan usar la menor cantidad posible de palabras que transmitan la información que queremos comunicar. Si estudiamos la cantidad de información de cada palabra desde el punto de vista de la teoría de la información veremos que responde a la ley de Zipf.

Esta explicación no aplica a otros casos como el diámetro de los cráteres lunares o la magnitud de los terremotos. En estos casos podríamos pensar en una explicación estadística. Si expresamos una distribución de probabilidades en función del ranking y aplicamos la expansión por Taylor de primer orden en todos los casos obtenemos la ley de Zipf, esto vale para la distribución normal, exponencial, y todas las demás (!). Sin embargo no todos las distribuciones del mundo real son Zipfianas.

En definitiva podemos pensar que una combinación de factores y que depende de como se relacionan los eventos entre sí determinan que una distribución responda a una ley de potencias y la ley de Zipf.

1.4.5 Resumen

Las leyes de potencias van a ser muy importantes en este texto ya que se pueden aplicar a muchos tipos de datos diferentes. Los dos ejemplos mas claros son la frecuencia de las palabras en textos con los que trataremos muy a menudo y

el grado de los nodos en una Red Social que es un tema que estudiaremos mas adelante.

Los conceptos mas importantes que deben quedar claros son:

1. La ley de Zipf, la ley de Pareto y las leyes de potencias son formas de visualizar el mismo fenómeno. Son la misma cosa.
2. Podemos identificar una ley de potencias cuando el plot en escala logarítmica de dos variables es una recta.
3. Las leyes de potencias solo tienen promedio si el exponente es mayor a 2
4. Las leyes de potencias tienen varianza infinita si el exponente es menor a 3
5. La mayoría de las leyes de potencias en el mundo real responden a un exponente entre 2 y 3 ya que cumplen la ley de Zipf.
6. No es correcto determinar el exponente ajustando una recta al plot en escala log-log
7. Ley de potencias=distribución; Pareto=Distribución acumulada; Zipf=Frecuencia en funcion del Ranking.

1.5 Algunos Elementos de Probabilidad y Estadística

En todo proceso de análisis de datos es necesario usar una cierta cantidad de conocimientos sobre estadística, en un extremo podríamos afirmar que "Data Science" es el nombre moderno que le damos a la estadística aplicada usando algoritmos especiales. Es conveniente que todos aquellos que quieran dedicarse a esta rama de la computación tengan una fuerte base de probabilidades y estadística, es imposible en este curso hacer un listado o repaso de todos los conceptos que serían útiles o necesarios por lo que vamos a concentrarnos en algunos temas curiosos y pocas veces mencionados que son importantes para el procesamiento de datos.

1.5.1 El Principio de Bonferroni

El principio de Bonferroni está directamente relacionado con la actividad mas importante de la estadística: contar. Podemos enunciar el principio de Bonferroni de la siguiente manera: "Es posible que ciertos datos aleatorios se confundan con aquellos que estamos buscando realmente". Una forma de entender esto es planteando la probabilidad de que cierto evento ocurra de forma aleatoria y contando la cantidad esperada de casos, si estos casos son muchos entonces los resultados de nuestro proceso pueden estar contaminados con una gran cantidad

de casos "ruidosos" que no responden a lo que realmente estamos buscando sino que son simplemente producto de la casualidad.

Veamos el siguiente ejemplo (Tomado del libro Mining Massive Datasets). Supongamos que queremos encontrar terroristas y que nuestra presunción es que estos se han reunido mas de una vez en hoteles diferentes. Para conocer la cantidad de ruido que podemos encontrar en nuestra pesquisa necesitamos calcular la probabilidad de que dos personas comunes visiten el mismo hotel en días diferentes. Para calcular esto a modo de ejemplo tenemos los siguientes datos:

- La población total es de 1000 millones de personas (10^9)
- Una persona normal visita un hotel una vez cada 100 días
- Estudiamos un período de tiempo total de 100 días
- Cada hotel puede alojar a 100 personas
- En total tenemos 100.000 hoteles

Notemos que 100.000 hoteles con capacidad para 100 personas cada uno implica que un total de 10^7 personas pueden alojarse al mismo tiempo y eso coincide con el 1% de 10^9 .

Calculemos primero la probabilidad de que dos personas visiten un hotel es $0.01^2 = 0.0001$ para que visiten el mismo hotel hay que dividir por la cantidad de hoteles es decir 100.000 por lo tanto la probabilidad de que dos personas cualesquiera visiten el mismo hotel el mismo día es 10^{-9} . Para que esto ocurra dos veces es decir en dos días diferentes elevamos esta probabilidad al cuadrado y nos da 10^{-18} .

Ahora tenemos que considerar la cantidad total de eventos posibles. La cantidad total de pares de personas es $\binom{10^9}{2} = 5 * 10^{17}$. El número total de pares de días es $\binom{1000}{2} = 5 * 10^5$. Estamos aproximando $\binom{n}{2} \approx n^2/2$ lo cual es válido cuando n es un número grande.

Por lo tanto el número total de sucesos es igual a la cantidad de pares de personas por la cantidad de pares de días por la probabilidad de que un par de personas visiten el mismo hotel en dos días diferentes es decir:

$$5 * 10^{17} * 5 * 10^5 * 10^{-18} = 250000$$

Esto quiere decir que si buscamos terroristas con este método vamos a tener por lo menos 250000 casos que son simplemente producto de la casualidad, esta es la cantidad de ruido que vamos a tener que filtrar para encontrar los sucesos que realmente nos interesan. La conclusión mas importante que podemos obtener de estudiar el principio de Bonferroni es que en un set de datos muy grande es probable que cualquier proceso de filtrado arroje una cantidad importante de resultados falso-positivos que luego tenemos que de alguna manera detectar y eliminar.

1.5.2 La Ecuación mas peligrosa de la historia

En esta sección vamos a presentar algo realmente escalofriante, una ecuación que a lo largo de la historia, desde tiempos muy antiguos hasta el día de hoy la mayoría de la gente desconoce. Esto no sería grave ya que existen millones de ecuaciones que la mayoría de la gente desconoce. Sin embargo el desconocimiento de esta ecuación ha causado verdaderos estragos económicos o de interpretación de datos. La ecuación en cuestión es la denominada ecuación de Moivre y sirve para calcular la desviación standard del promedio de un conjunto de muestras.

$$\sigma(\bar{x}) = \frac{\sigma}{\sqrt{n}}$$

Lo que esta ecuación nos dice es que la desviación del promedio es igual a la desviación de la muestra dividido la raíz del tamaño de la misma.

La primera aparición de esta ecuación en la historia ocurre en el año 1150. En esa época se fabricaban y acuñaban monedas de oro de acuerdo a una cierta especificación que determinaba su valor, por ejemplo cada moneda debía pesar 28 gramos +/- 0.6 gramos para ser considerada legal. Luego de la fabricación de las monedas resultaba muy costoso pesar cada moneda individualmente por lo que se pesaban en lotes de 100 monedas y como cada una podía variar 0.6 gramos se aceptaba que las 100 monedas tuvieran una variación de +/- 60 gramos. Simple, efectivo y terriblemente equivocado.

Algunos orfebres notaron que algo no estaba bien y empezaron a separar las monedas en lotes en las cuales una de las monedas era del doble del peso especificado, al pesar las cien monedas juntas esta moneda pasaba desapercibida y el orfebre podía luego fundirla y quedarse una moneda extra por cada lote. Este ingenioso proceso de trampa ocasionó perdidas millonarias. La culpa por supuesto es de la ecuación de Moivre ya que para un lote de 100 monedas la variabilidad aceptable no tendría que ser 100 por 0.6 sino 10 por 0.6 es decir 6 gramos. Así comenzó el largo camino de la ecuación de Moivre causando todo tipo de problemas.

A fines de la década de los 90 la fundación de Bill y Melinda Gates realizó un estudio sobre el rendimiento académico de las escuelas en EEUU. El resultado mostró que las escuelas con mejor rendimiento académico eran en general escuelas con pocos alumnos. Con este resultado la fundación Gates comenzó un plan para dividir las escuelas mas grandes en escuelas mas pequeñas invirtiendo una considerable cantidad de dinero en esta tarea. El argumento era muy simple: en una escuela pequeña el trato mas personalizado permitía que los alumnos aprendan mas y logren mejores resultados. Por supuesto todo esto no era cierto sino el resultado de una vez mas ignorar la ecuación de Moivre. Las escuelas con menor cantidad de alumnos son las que presentan mayor variabilidad, unos pocos alumnos con buen desempeño en una escuela de pocos alumnos dispara el ranking de la escuela a los primeros puestos. Esto no quiere decir que las escuelas mas chicas sean mejores, el ranking de las peores escuelas también muestra una preponderancia de escuelas con pocos alumnos.

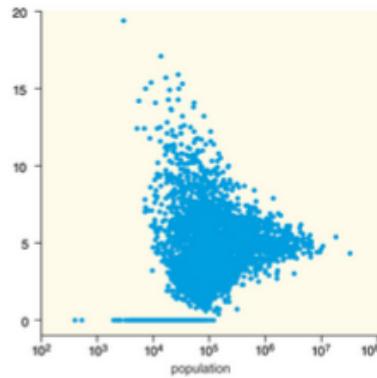


Figure 1.19: Variabilidad en función de la población

El gráfico muestra la variabilidad en función de la población la forma triangular es típica, los casos con pocas muestras tienen una variabilidad mucho mayor que otros con una población mucho mayor. Eventualmente alguien hizo los estudios adecuados y estos arrojaron que por el contrario las escuelas con mayor cantidad de alumnos tenían un mejor rendimiento académico, esto se puede explicar en función de tener una mayor cantidad de docentes y de que los mejores docentes en general quieren ir a las escuelas mas grandes.

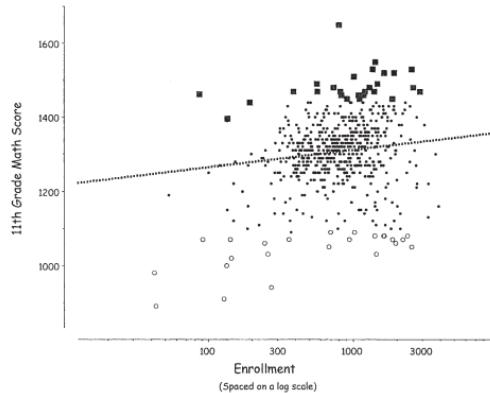


Figure 1.20: Rendimiento de alumnos del 11vo grado

Finalmente la fundación Gates dio marcha atrás y volvió a reintegrar las escuelas con muchos alumnos que habían sido divididas en varias escuelas menores. En total se gastaron miles de millones de dólares y solamente por el desconocimiento de una simple ecuación.

El ataque de Moivre nunca ha cesado, hoy en día se publican todo el tiempo estudios en los cuales se evidencia el desconocimiento de la ecuación. Recientemente alguien publicó una lista de las ciudades más seguras en EEUU y, como es de esperar, el listado estaba dominado por ciudades con menos de 500.000 habitantes. Lamentablemente es probable que esto nunca tenga solución.

1.5.3 Frecuentistas vs Bayesianos

De la sección anterior hemos aprendido que cuando tenemos pocos datos la variabilidad de los mismos es mayor que cuando tenemos muchos datos, en pocas palabras cuantos mas datos tenemos mas confiable es nuestra estadística y comparar estadísticas de datos con diferentes espacios muestrales suele ser una idea peligrosa. Esto nos lleva a introducir las dos grandes corrientes de pensamiento dentro de la estadística que son la corriente frecuentista y la corriente bayesiana.

La diferencia básica entre el punto de vista frecuentista y el punto de vista bayesiano radica en la información necesaria para obtener datos estadísticos. Para un frecuentista las estadísticas surgen de los datos mismos mientras que para un bayesiano las estadísticas surgen de la combinación de los datos y un cierto conocimiento a-priori sobre los mismos.

Para un frecuentista $5/10$ y $1000/2000$ son lo mismo: 0.5 para un bayesiano son cosas muy diferentes...

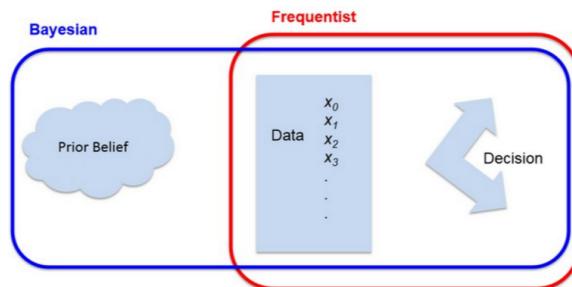


Figure 1.21: Frecuentistas vs Bayesianos

Consideremos un ejemplo muy simple: en el primer partido de la temporada un jugador de basket intenta tres triples y logra meter los tres. ¿Es lógico suponer que su efectividad en triples a lo largo de la temporada será del 100%?. Para un frecuentista la respuesta es "sí" ya que con los datos que tenemos el porcentaje de triples del jugador es $3/3$ y por lo tanto no tenemos mas que suponer que su porcentaje durante la temporada será el mismo. Para un bayesiano esto no es así ya que tenemos que tener en cuenta el porcentaje de triples a-priori de un jugador de basket, supongamos que el porcentaje promedio de triples de todos los jugadores que juegan en la misma posición es de un 52%, de acuerdo a la filosofía bayesiana la predicción para el jugador durante la temporada será

de 52% mas un pequeño epsilon por la evidencia que tenemos al verlo meter los primeros 3 de 3 intentos en su primer partido.

Este sencillo ejemplo nos enseña varias cosas: de acuerdo a la filosofía bayesiana la probabilidad de un evento está dada por la probabilidad a-priori del mismo y la evidencia que tenemos (observaciones), cuanta mas evidencia tenemos mas podemos desviarnos de nuestro conocimiento a-priori.

Veamos un segundo ejemplo: tenemos un test que tiene un 99% de efectividad para detectar si alguien es pelirrojo. Si el test nos da positivo, ¿cuál es la probabilidad de que seamos pelirrojos?. Para un frecuentista la probabilidad es del 99% y no hay mas discusión al respecto y es probable que ésta sea la respuesta mas popular para el público en general, al fin y al cabo el test tiene 99% de efectividad. Para un bayesiano es necesario saber cuál es la probabilidad a-priori, es decir la probabilidad de que alguien sea pelirrojo independientemente del test. Supongamos que se sabe a-priori que uno de cada 100 individuos es pelirrojo. Con esa información podemos pensar la respuesta bayesiana a nuestra pregunta. Tenemos 100 personas, una de ellas pelirroja, les hacemos el test y como el porcentaje de eficiencia es del 99% sabemos que le va a dar mal a una persona. Supongamos que al que es pelirrojo el test le da bien, es decir que le dice "usted es pelirrojo" de los otros 99 hay 1 al cual el test le va a decir incorrectamente que es pelirrojo. En definitiva tenemos 2 respuestas de tipo "usted es pelirrojo" y solo 1 de esas 2 personas realmente lo es, por lo tanto la probabilidad de ser pelirrojo si el test nos da positivo es del 50%, bayesian magic.

Formalmente acabamos de aplicar de forma intuitiva el teorema de Bayes:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \quad (1.6)$$

Este teorema rige todo el proceso de inferencia bayesiana y de alguna forma se encuentra hard-codeado en el cerebro humano ya que incluso aquellos que jamás han oído hablar del mismo lo pueden aplicar cuando la situación es evidente, por ejemplo en el caso de nuestro jugador de basket. Verifiquemos si el teorema se cumple para nuestro test de pelirrojidad.

$P(A|B)$ es la probabilidad de ser pelirrojos si el test nos da positivo que es lo que queremos averiguar.

$P(B|A)$ es la probabilidad de que el test nos de positivo si somos pelirrojos es decir 0.99

$P(A)$ es la probabilidad de ser pelirrojos en general: 0.01

$P(B)$ es la probabilidad de que el test nos de positivo en general intuitivamente podemos razonar que es 2/100 ya que le dará positivo al que es pelirrojo y a uno que no es dado que el test no es infalible. Esto lo podemos comprobar calculando:

$$P(B) = P(B|A)P(A) * P(B|not A) P(not A)$$

$P(B|not A)$ es la probabilidad de que el test de positivo si no somos pelirrojos: 0.01 $P(not A)$ es la probabilidad de no ser pelirrojos es decir 0.99

$P(B) = 0.99 * 0.01 + 0.01 * 0.99 = 0.0198$ que es el 2/100 que habíamos calculado.

Entonces:

$$P(\text{pelirrojo}+) = P(\text{test+—pelirrojo})P(\text{pelirrojo})/P(\text{test+}) = 0.99 * 0.01 / 0.02 = 0.5$$

De los dos ejemplos que hemos visto podemos concluir que en una persona promedio el cerebro a veces funciona de forma frecuentista y a veces de forma bayesiana, en el caso del basketbolista casi todos pensamos de forma bayesiana pero en el caso del test de pelirrojidad(?) la gran mayoría de la población comparte el pensamiento frecuentista.

La aproximación al pensamiento bayesiano automáticamente nos lleva a pensar varios problemas de una forma diferente. Veamos un tercer caso. Le tomamos a un grupo de personas un examen de 10 preguntas de cultura general y obtenemos los resultados. Ahora al mismo grupo de personas le tomamos un segundo examen de 10 preguntas de cultura general pero diferente al primero. ¿Cuál sería una buena forma de estimar el resultado para cada persona?

La aproximación frecuentista al problema es estimar para cada persona el mismo resultado que el examen anterior, si alguien respondió bien 8 preguntas estimamos que va a volver a responder bien 8 preguntas. Esto es lógico pero desde el punto de vista bayesiano es incorrecto. Para una respuesta bayesiana el resultado del examen anterior es la "evidencia", necesitamos además un cierto conocimiento a-priori y a falta de otra cosa podemos usar el promedio de los resultados de todos los exámenes (μ). La predicción entonces será de la forma $e2 = \lambda\mu + (1 - \lambda)e1$ siendo λ un parámetro que regula el peso de nuestro conocimiento a-priori y nuestra evidencia. Resulta ser que con $\lambda = 0.5$ la predicción del resultado del segundo examen es mejor que la que haríamos con la filosofía frecuentista.

Estimación Bayesiana con la Distribución Beta

Para formalizar un poco la forma de usar la filosofía bayesiana vamos a ver un método de partiendo de una cierta distribución de probabilidades a-priori ir actualizándola en función de la evidencia que observamos. Para hacer esto vamos a usar la distribución Beta. La distribución beta es frecuentemente ignorada o descripta de formas muy complejas sin necesidad, lo que la distribución en realidad mide es una distribución de probabilidades y por eso es muy interesante desde el punto de vista bayesiano.

A modo de ejemplo consideremos el problema de estimar el porcentaje de triples para la presente temporada para un cierto jugador. Sabemos que para la posición en la que el jugador juega el porcentaje promedio de triples es 0.38 con una desviación standard de 0.11 (varianza = 0.0121). La distribución beta tiene dos parámetros: α y β que tenemos que encontrar antes de poder usarla.

En primer lugar la varianza se puede expresar en función del promedio como:

$$\sigma^2 = \frac{\mu(1 - \mu)}{\alpha + \beta + 1}$$

Si queremos $\mu = 0.38$ y $\sigma^2 = 0.0121$ entonces calculamos:

$$\alpha + \beta = \frac{\mu(1 - \mu)}{\sigma^2} - 1 = \frac{.38(1 - .38)}{.0121} - 1 = 18.47$$

Y conociendo el total para $\alpha + \beta$

$$\alpha = \mu(\alpha + \beta) = .38(18.47) = 7.01$$

$$\beta = (1 - \mu)(\alpha + \beta) = 11.45$$

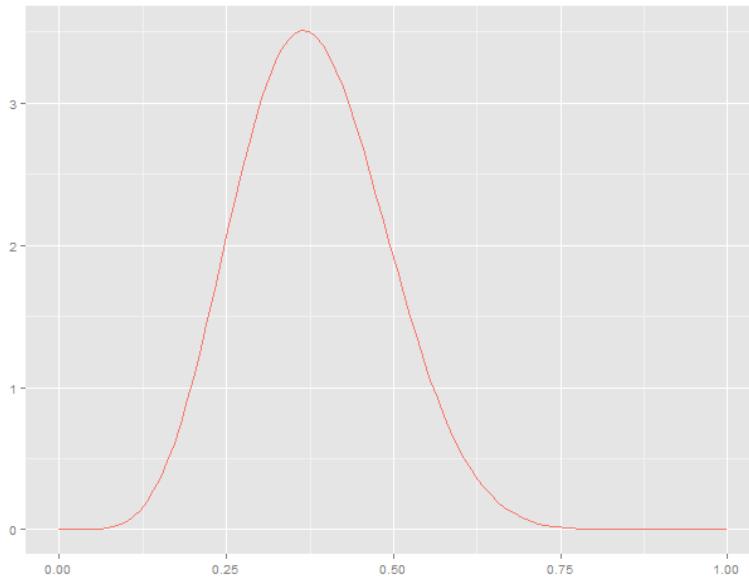


Figure 1.22: Distribución Beta con $\alpha = 7.01$ y $\beta = 11.45$

Como podemos ver la distribución beta cumple razonablemente con lo que necesitamos, tiene promedio 0.38 y desviación standard 0.11. El promedio de la distribución beta es $\frac{\alpha}{\alpha+\beta}$.

El gran truco de la distribución beta es que resulta sumamente sencillo actualizarla en función de nueva evidencia.

$$\beta_2 = \beta_1(\alpha + \text{hits}, \beta + \text{misses})$$

Suponiendo que nuestro jugador mete 3 triples seguidos nuestra nueva distribución beta es:

$$\text{beta}_2(7.01 + 3, 11.45 + 0)$$

Si calculamos el promedio ahora tenemos

$$10.01/10.01 + 11.34 = 0.46$$

notemos que nuestra predicción para el jugador en la temporada ahora es 0.46, bastante mejor que 0.38 pero lejos de un 100% que sería la predicción frecuentista.

Supongamos que luego de varios partidos nuestro jugador mete 21 de 50 triples. Ahora el promedio es:

$$21 + 7.01 / (11.45 + 7.01 + 50) = 0.409$$

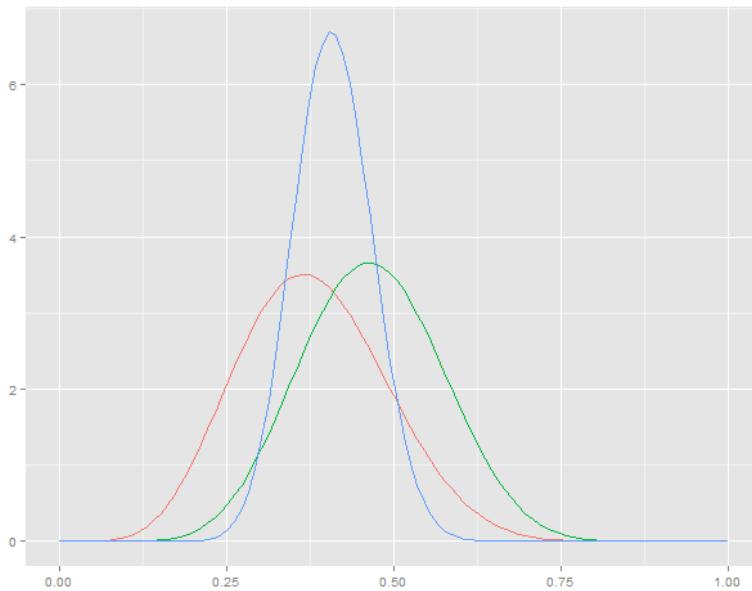


Figure 1.23: Las tres distribuciones beta

La figura 1.23 muestra las tres distribuciones. La roja es nuestra distribución a-priori con promedio 0.38, la verde es la distribución luego de 3 triples consecutivos, tiene promedio 0.46 la azul es la distribución luego de ver al jugador intentar 50 triples con los 21 aciertos mencionados. Observemos que a medida que agregamos evidencia la desviación standard disminuye es decir que cada vez estamos mas convencidos de cual será el porcentaje de triples del jugador durante la temporada.

La distribución beta simplifica enormemente el análisis bayesiano de las probabilidades en base a una cierta distribución a-priori y la evidencia que vamos observando. Es tan simple que muchas veces nadie cree que la estimación puede surgir de una cuenta tan sencilla como agregar los casos exitosos a α y los casos no exitosos a β y calcular $\alpha / (\alpha + \beta)$ para estimar el promedio.

```
// We hired a Data Scientist to analyze our Big Data
// and all we got was this lousy line of code.
float estimate = (successes + 78.7) / (total + 303.5);
```

La filosofía bayesiana da lugar al algoritmo de clasificación *naive bayes* que veremos mas adelante y a las redes bayesianas, y de acuerdo a quienes profesan esta corriente, es la explicación para todo lo que ocurre en el universo!

1.5.4 La Única Ley sin Explicación

Es muy difícil explicar una ley que no se cumple siempre pero casi y que no tiene ningún tipo de demostración aceptada pero que al mismo tiempo ha llegado a aceptarse como prueba en pericias judiciales. Estamos hablando de la ley de Benford. Esta ley postula lo siguiente: "En un conjunto de datos numéricos la distribución del primer dígito de cada número no es uniforme sino que los dígitos mas pequeños aparecen en mayor proporción que los mas grandes" En otras palabras estamos diciendo que si tenemos un conjunto de datos numéricos y contamos la frecuencia del primer dígito de cada número vamos a encontrar que el dígito 1 es mucho mas frecuente que el 2, este es mas frecuente que el 3 y así sucesivamente.

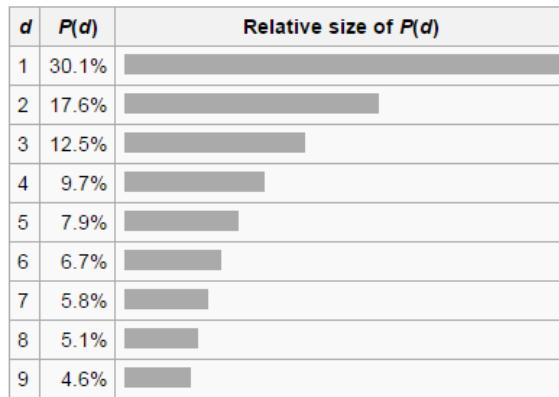


Figure 1.24: Distribución de los dígitos según la ley de Benford

Como vemos en el gráfico la distribución es logarítmica: un 30% de los números en nuestro set de datos empezarán con unos, un 17.6% con 2 y así sucesivamente hasta el 9 que debería encabezar un 4.6% de los números. Esta ley parece no tener sentido alguno sin embargo es válida para una enorme cantidad de sets de datos, por ejemplo la altura de los edificios más altos del mundo, la distancia entre el Sol y las estrellas más cercanas, etc.

La ley de Benford suele cumplirse con tanta regularidad que se ha aceptado como pericia para demostrar fraudes cuando la presentación de ciertos valores numéricos por una de las partes no ajusta a la distribución esperada de acuerdo a la ley de Benford. Para transacciones financieras la ley de Benford se cumple con una efectividad casi asombrosa.

En la figura 1.25 a la izquierda vemos como la información financiera pública ajusta casi de forma perfecta a la ley de Benford, en la derecha vemos los

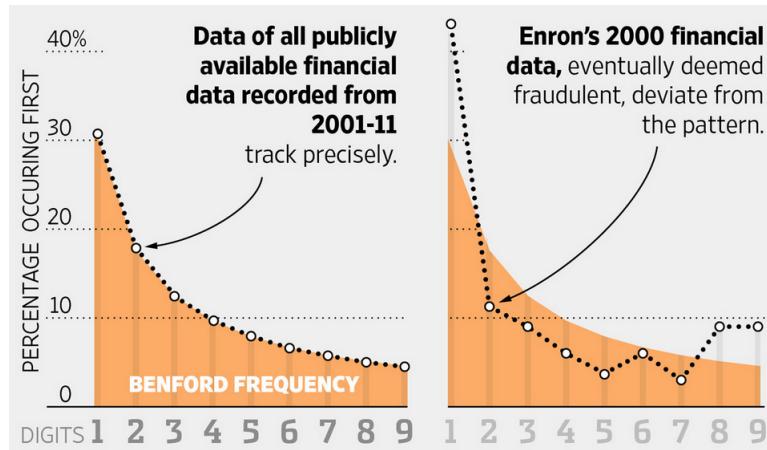


Figure 1.25: La ley de Benford en transacciones financieras públicas vs los datos entregados por Enron.

valores para los informes presentados en 2000 por Enron y que eventualmente se encontró de carácter fraudulento, ¡los números no coinciden!

Hay varios intentos de explicar la ley de Benford pero ninguno ha sido aceptado como lo suficientemente riguroso, aunque parezca mentira esta ley no tiene explicación ni demostración, simplemente se cumple.

1.5.5 Skewness

El fenómeno conocido como "skewness" se da cuando la distribución de las clases en nuestro set de datos está muy desbalanceada. Consideremos el ejemplo típico de una base de datos con emails rotulados como spam o no-spam que queremos usar como set de entrenamiento para un clasificador que nos permita predecir si un email no rotulado es spam o no. En el set de entrenamiento vamos a tener muchos más emails rotulados como no-spam que como spam ya que afortunadamente el spam es sólo un pequeño porcentaje del total del volumen de emails que circulan diariamente. Por ejemplo podríamos tener un set de entrenamiento en el cual un 1% de los mensajes están rotulados como spam.

Este desbalanceo es peligroso ya que puede afectar seriamente el funcionamiento de nuestros algoritmos. Como ejemplo consideremos un algoritmo que simplemente predice "no-spam" para cualquier texto que se le presente. De acuerdo con nuestro set de entrenamiento este algoritmo tendrá una precisión del 99% y sin embargo no hace absolutamente nada.

La performance de una predicción muy simple la llamaremos "baseline", cuando el set de datos está muy desbalanceado el baseline suele fijar la vara muy alta por lo que hay que tomar con mucho cuidado el resultado de nuestros

algoritmos.

Hay varias formas de combatir el skewness, una es realizar un muestreo estratificado del set de datos para asegurarnos que tenemos un cierto porcentaje de datos de cada clase, esto va a ayudar a los algoritmos a distinguir mejor entre las clases que existen sin introducirles el bias de saber que una de las clases es mucho mas importante que la otra.

En otros casos es posible asignar pesos a los errores de clasificación y podemos decir que el error de clasificar un mail normal como spam es mucho mas alto que el de clasificar un spam como normal, de esta forma el algoritmo va a tender a tener mas cuidado en la clasificación que realice. Este tema aparece muy frecuentemente y la forma de solucionarlo es dependiente del algoritmo que usemos por lo que será siempre un factor fundamental a considerar en el análisis exploratorio de nuestros datos.

1.6 Algoritmos Aleatorizados

Los algoritmos aleatorizados son aquellos que en algún momento toman alguna decisión al azar. Podemos dividirlos en dos grandes familias:

Algoritmos tipo Montecarlo Estos algoritmos siempre terminan en una cantidad de tiempo acotada pero pueden no encontrar la solución óptima. En su forma mas básica podemos decir que prueban soluciones al azar y se quedan finalmente con la que optimiza una cierta función, ya sea maximizándola o minimizándola para el conjunto de soluciones probadas.

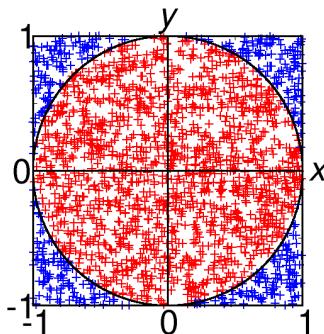
Algoritmos tipo Las Vegas Estos algoritmos siempre encuentran la solución óptima pero, con una cierta probabilidad, pueden tardar mucho tiempo en encontrarla.

A modo de ejemplo veamos un algoritmo de tipo Montecarlo para calcular el valor de π . La idea es muy simple: vamos a tener un cuadrado de lado $2r$ y un círculo inscripto dentro del mismo de radio r . Nuestro algoritmo va a generar puntos al azar dentro del cuadrado y va a verificar si caen dentro o fuera del círculo, sabiendo que si caen dentro entonces $x^2 + y^2 < r^2$. Luego de simular millones de puntos tenemos la cantidad de puntos totales y la cantidad de estos puntos que cayeron dentro del círculo y a partir de estas cantidades podemos estimar el valor de π .

El área del círculo es $\pi * r^2$ y el área del cuadrado es $(2r)^2$ por lo tanto si dividimos el área del círculo por el área del cuadrado obtenemos:

$$\frac{\pi * r^2}{(2r)^2} = \frac{\pi}{4}$$

Por lo tanto si elegimos N puntos al azar dentro del cuadrado entonces $M = N * \pi/4$ van a caer dentro del círculo. Como nuestra simulación cuenta la cantidad M y N podemos calcular

Figure 1.26: Calculando π mediante Montecarlo.

$$\pi \approx \frac{4M}{N}$$

Algorithm 1: Algoritmo tipo Montecarlo para calcular pi

Data: r: circle radius, N:number of iterations

Result: pi

```

1 M=0;
2 N=0;
3 for i in (1..N) do
4   | x = rand(0,2r);
5   | y = rand(0,2r);
6   | if  $x^2 + y^2 < r^2$  then
7   |   | M= M +1
8 pi = 4M/N;
```

Consideremos ahora el problema de calcular el promedio en bytes del tamaño de todas las páginas web. Claramente no es factible simplemente tomar todas las páginas web y calcular el promedio de su tamaño en bytes. Un algoritmo de tipo Montecarlo puede aproximar esta cantidad simplemente tomando una cierta cantidad fija de páginas al azar, por ejemplo un millón y calculando su promedio. De acuerdo a la ley de los grandes números cuanto mayor sea el tamaño de la muestra que tomamos mas cerca va a estar nuestra estimación de converger a la solución verdadera.

1.6.1 Algoritmo de Fermat

El algoritmo de Fermat es un algoritmo de tipo Montecarlo muy simple para determinar si un número es primo, está basado en el pequeño teorema de Fermat que dice:

Theorem 1 (Fermat) *si p es primo entonces para*

$$0 < a < p$$

$$a^{p-1} = 1 \pmod{p}$$

El algoritmo entonces simplemente genera valores al azar para a y si todos los valores probados cumplen con el teorema entonces el número es primo.

Algorithm 2: Algoritmo de Fermat

Data: $p, \text{iterations}$
Result: composite if n is composite, otherwise probably prime

```

1 for i in (1..iterations) do
2   if  $a^{p-1} \bmod p \neq 1$  then
3     return composite;
4 return probably prime;
```

1.6.2 Algoritmo de Miller-Rabin

El algoritmo de MillerRabin es un algoritmo tipo Montecarlo para probar si un número es primo o no, es mas eficiente que el algoritmo de Fermat en el sentido de que es menos probable que un número que no es primo pase el test.

1.6.3 Random Walks

Un Random Walk es un recorrido al azar, por ejemplo podemos generar un random walk al azar en dos dimensiones comenzando en una cierta coordenada al azar (X, Y) y luego desplazándonos a $X+1, X-1, Y+1$ o $Y-1$.

Los Random Walks son extremadamente útiles para simular todo tipo de procesos y estimar luego parámetros que son difíciles de calcular analíticamente o que no sabemos calcular.

1.6.4 Markov Chains

Una cadena o proceso de Markov es un grafo dirigido en donde cada nodo representa un estado y cada arista representa un transición de un estado a otro. Cada arista lleva un rótulo indicando su probabilidad. La suma de todas las aristas salientes de un nodo debe ser 1.

Algorithm 3: Algoritmo de Miller-Rabin

Data: $n > 2$, an odd integer to be tested for primality;
 k , a parameter that determines the accuracy of the test;

Result: Output: composite if n is composite, otherwise probably prime

```

1 write  $n - 1$  as  $2^s * d$  with  $d$  odd by factoring powers of 2 from  $n - 1$ ;
2 for  $i$  in  $(1..k)$  do
3    $\downarrow$  p
4   pick a randomly in the range  $[2, n - 1]$ ;
5    $x = a * d \bmod n$ ;
6   if  $x = 1$  or  $x = n - 1$  then
7      $\downarrow$  do next LOOP
8   for  $r=1 .. s-1$  do
9      $x = x^2 \bmod n$  if  $x = 1$  then
10       $\downarrow$  return composite
11      if  $x = n-1$  then
12         $\downarrow$  do next LOOP
13  return composite;
14  return probably prime;
```

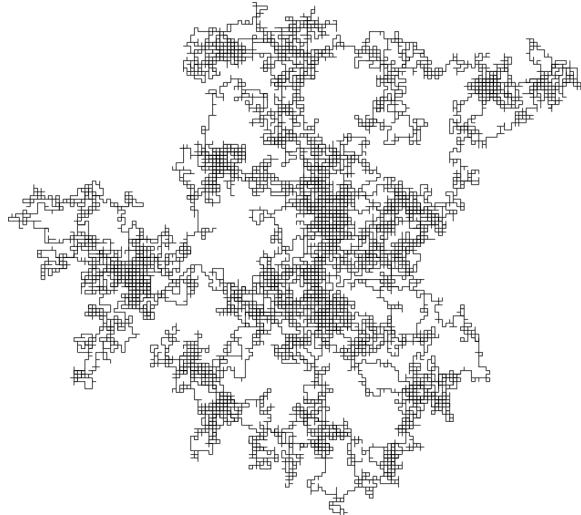


Figure 1.27: Random Walk en 2d (25000 iteraciones).

Las cadenas de Markov pueden usarse para modelar todo tipo de procesos, por ejemplo podemos modelar información meteorológica usando como nodos los diferentes estados del tiempo (soleado, parcialmente nublado, nublado, lluvia,

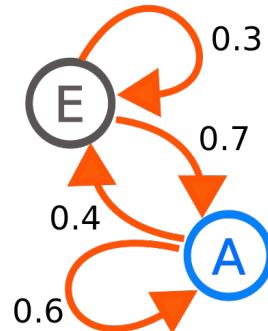


Figure 1.28: Cadenas de Markov.

tormenta) y cada arista representa la probabilidad de pasar de un estado a otro.

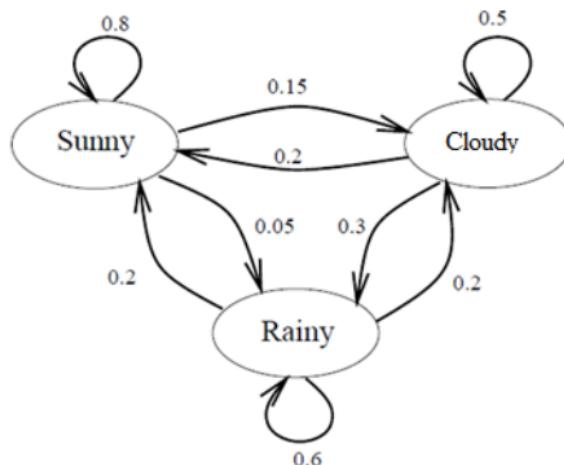


Figure 1.29: Proceso de Markov para el pronóstico del tiempo.

Como podemos ver si el día está soleado lo mas probable es que mañana también esté soleado y lo menos probable es que llueva. Hay muchísimas cosas que podemos hacer con una cadena de Markov de forma analítica pero es destacable que sin tener absolutamente ningún conocimiento sobre procesos de Markov podemos responder prácticamente cualquier pregunta mediante una simulación de tipo Montecarlo.

Como ejemplo supongamos que queremos saber cuál es la probabilidad de que dentro de un mes tengamos tres días soleados. Realizamos un Random

Walk comenzando en cualquier estado al azar y probabilísticamente vamos moviéndonos de un estado a otro hasta simular los 30 días, luego simplemente contamos cuantas veces aparecen tres estados "sunny" consecutivos, esto lo podemos repetir n veces y calcular un promedio para mayor efectividad siempre amparados en el poder de la ley de los grandes números.

Realizar la simulación es muy simple, por cada estado(nodo) generamos un número random entre 0 y 1 y nos fijamos a que transición corresponde. Por ejemplo supongamos que estamos en "sunny" tenemos probabilidad 0.05 de ir a "rainy", 0.15 a "cloudy" y 0.8 de seguir en "sunny" generamos un número entre 0 y 1 al azar y si el número es menor o igual a 0.8 entonces el próximo estado es "sunny" si el número es mayor a 0.8 y menor a 0.95 el próximo estado es "cloudy" y si el número es mayor a 0.95 el próximo estado es "rainy" como podemos ver esto es acorde con la probabilidad de cada arista en nuestro grafo.

Modelando Lenguajes como Procesos de Markov

Una de las aplicaciones mas sorprendentes de un proceso de Markov es el modelado de un lenguaje. Podemos procesar una gran cantidad de textos y contabilizar la cantidad de veces que un carácter aparece luego de otro, de esta forma una vez procesados todos los textos podemos tener un grafo en el cual cada nodo representa un carácter y las transiciones representan la probabilidad de que un carácter aparezca luego de otro. Un random walk en este grafo genera texto al azar imitando las propiedades de los textos que procesamos. El modelo puede mejorarse si en lugar de caracteres individuales tomamos grupos de n caracteres que llamaremos n -gramas por ejemplo podemos registrar por cada 3 caracteres la probabilidad del próximo carácter, en nuestro grafo cada nodo representa ahora cuáles son los n caracteres anteriores y las transiciones representan cuál será la probabilidad del próximo carácter. A modo de ejemplo procesamos todas las obras de Shakespeare y luego usando $n = 10$ simulamos un random walk sobre el grafo el resultado es el siguiente:

First Citizen:

Nay, then, that was hers,
It speaks against your other service:
But since the
youth of the circumstance be spoken:
Your uncle and one Baptista's daughter.

SEBASTIAN:

Do I stand till the break off.

BIRON:

Hide thy head.

VENTIDIUS:

He purposeth to Athens: whither, with the vow

I made to handle you.

FALSTAFF:

My good knave.

MALVOLIO:

Sad, lady! I could be forgiven you, you're welcome. Give ear, sir, my doublet and hose

Second Gentleman:

Who may that she confess it is my lord enraged and forestalled ere we come to be a man

APEMANTUS:

Ho, ho! I laugh to see your beard!

Como podemos apreciar el texto no solamente tiene sentido sino que luce como Shakespeare aunque haya sido generado completamente al azar por un simple programa. Esto muestra el enorme poder de los procesos de Markov para entender una secuencia de eventos y la increíble efectividad de un simple Random Walk tipo Montecarlo como proceso generativo. Es por esto que muchos compresores de texto utilizan modelos de Markov para modelar el archivo a comprimir logrando niveles de compresión excelentes.

1.6.5 Hill Climbing

Hill Climbing es una familia de algoritmos con un principio muy simple: comenzar con una solución al azar, proponer un cambio y aceptarlo solamente si el cambio mejora la solución. Podemos decir que se trata de una meta-heurística ya que se trata de un algoritmo para construir otros algoritmos.

Algorithm 4: Hill Climbing

```

1 create a random solution s;
2 for i in (1..iterations) do
3   propose an alternative solution (s');
4   if s' better than s then
5     s = s';
6 return s;
```

Para transformar una solución en otra se tiene que realizar lo que se denomina un "cambio local" es decir un pequeño cambio en la solución. Por ejemplo supongamos que queremos minimizar la función $2x^2 - 3x + 2$ podemos comenzar con un cierto valor de x al azar generar un ϵ al azar positivo o negativo y proponer la solución $x + \epsilon$ aceptando este valor solamente si la función ha disminuido entre una solución y la otra. Eventualmente este simple proceso de Hill-Climbing va converger al mínimo de la función.

El proceso de Hill-Climbing no es ni mas ni menos que un algoritmo "greedy" y no funciona si la función que queremos maximizar o minimizar tiene máximos o mínimos locales ya que una vez que comenzamos en una cierta solución al azar vamos a poder llegar únicamente al máximo o mínimo local mas cercano a la misma. Para evitar quedar atrapados en extremos locales usamos una versión mejorada de Hill-Climbing que se denomina algoritmo de Metrópolis-Hastings.

1.6.6 El Algoritmo Metrópolis-Hastings

La variante que introducimos en el algoritmo de Metrópolis-Hastings es aceptar una solución que empeora nuestro resultado con una pequeña probabilidad.

Algorithm 5: Metrópolis-Hastings

```

1 create a random solution s;
2 for  $i$  in  $(1..iterations)$  do
3   propose an alternative solution  $(s')$ ;
4   if  $s'$  better than  $s$  or if  $random(0, 1) \leq p$  (a small number) then
5      $s = s'$ ;
6 return  $s$ ;

```

La heurística de Metrópolis-Hastings mejora el resultado de Hill-Climbing para funciones que tienen mínimos y máximos locales pero en general no se usa sino como parte de un algoritmo mas avanzado denominado "Simulated Annealing"

1.6.7 Simulated Annealing

Simulated Annealing (SA) es un meta-algoritmo extremadamente poderoso que se usa para aproximar soluciones a todo tipo de problemas muy difíciles de solucionar por otros métodos, es uno de los algoritmos mas usados para aproximar problemas que se sabe son NP-Complejos. La idea es partir del algoritmo Metrópolis-Hastings e ir variando la probabilidad de aceptar una solución que empeora nuestro resultado. Al principio cuando recién empezamos a explorar el espacio de soluciones queremos aceptar soluciones que no mejoran el resultado final con una cierta probabilidad " p_1 " luego se supone que nuestra solución va mejorando y queremos que esta probabilidad vaya disminuyendo hasta llegar a ser prácticamente cero en las últimas iteraciones. Esto lo logramos variando un parámetro que llamaremos la "temperatura" del algoritmo.

Supongamos que estamos minimizando una cierta función $f(s)$; al igual que siempre vamos a comenzar con una cierta solución al azar s y proponer un pequeño cambio para generar la solución s' .

Tenemos que definir entonces una función que nos diga la probabilidad de aceptar a esta nueva solución $P(s, s', T)$ la probabilidad de aceptar s' depende de s, s' y la temperatura T . Esta función nos debe devolver una probabilidad

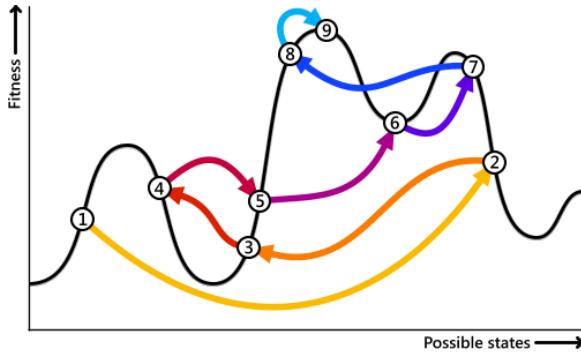


Figure 1.30: Ejemplo de Simulated Annealing en 9 iteraciones.

siempre, incluso cuando s' es mayor a s es decir una solución peor. Además queremos que cuando $T \rightarrow 0$, P tienda a cero. Si $s' > s$ cuando $T = 0$ el algoritmo se convierte en Hill-Climbing aceptando únicamente una solución alternativa si mejora el resultado actual. Una posible forma de definir la función de probabilidad P es:

$$P(s, s', T) = \begin{cases} 1 : s' < s \\ \exp(-(s' - s)/T) : \text{otherwise} \end{cases}$$

El algoritmo de Simulated Annealing quedaría entonces definido de la siguiente forma:

Algorithm 6: Simulated Annealing

```

1 create a random solution s;
2 for T in Tmax..0 step tstep do
3   | propose an alternative solution s';
4   | if random(0,1) ≤ P(s, s', T) then
5   |   | s = s';
6 return s;
```

La temperatura inicial T_{max} y el grado en el cual vamos disminuyendo la temperatura $tstep$ determinan la cantidad de iteraciones que vamos a realizar. Alternativamente podemos usar como parámetro la cantidad de iteraciones y T_{max} y calcular cuál es el step para ir desde T_{max} a 0 en dicha cantidad de iteraciones.

Simulated Annealing es un algoritmo muy importante ya que permite optimizar todo tipo de problemas conociendo únicamente como generar una solución alternativa a partir de cualquier otra solución (local change). Se usa para la aproximación de problemas como el problema del viajante y otros tipos de problemas muy complejos para los cuales no existe un método eficiente para llegar

a la solución óptima. Es crítico en estos algoritmos determinar los parámetros correctos para converger a la mejor solución posible, es común ir probando diferentes valores de temperatura y cantidad de iteraciones hasta llegar a la mejor solución posible.

1.7 MCMC

Vamos a empezar esta sección con un curioso ejemplo. Supongamos que queremos encuestar a la población con respecto a una pregunta en la cual suponemos que la mayoría de la gente va a mentir. Por ejemplo ¿Le gusta Arjona?. Vamos a proponer un algoritmo para poder realizar la encuesta:

1. Cada persona tira una moneda.
2. Si sale cara la persona responde honestamente.
3. Si sale seca la persona lanza una segunda moneda.
4. Si la segunda moneda es cara entonces responde "Si".
5. Si la segunda moneda es seca entonces responde "No"

Con este sencillo algoritmo nadie necesita mentir ya que es imposible discernir individualmente si una respuesta de "Si" fue honesta o simplemente el resultado de la segunda moneda. Es decir que podemos confiar en los resultados.

El problema es que nuestros resultados tienen ahora ruido que nosotros mismos hemos introducido intencionalmente. Supongamos que 100 personas responden usando el algoritmo propuesto y obtenemos 35 respuestas "Si".

De nuestras 100 personas podemos esperar que 50 hayan sacado cara en la primera moneda y hayan respondido honestamente. De los restantes 50 podemos esperar 25 respuestas "Si" y 25 respuestas "No". Por lo tanto de los otros 50 tiene que haber 10 respuestas afirmativas y eso nos permitiría afirmar que la probabilidad de que a una persona le guste Arjona es $10/50 = 0.2$.

Podemos ver que nuestro algoritmo ha disminuido el espacio muestral a cambio de privacidad en las respuestas, pero a menor espacio muestral tenemos mayor incertidumbre, ciertamente sería un error afirmar categóricamente que al 20% de la población le gusta Arjona (el cielo se apiade).

Lo que nos interesa es la *distribución* de probabilidad de $p(\text{arjona})$, en el enfoque Bayesiano queremos averiguar la distribución a-priori. El algoritmo que nos permite hacer esto es MCMC (Markov Chain Monte-Carlo).

Para entender MCMC pensemos una sencilla solución a nuestro problema. Podemos tirar un valor al azar para $p(\text{arjona})$ y simular 100 experimentos usando este valor, es decir 100 simulaciones de nuestro algoritmo con las dos monedas. Si como resultado obtenemos 35 respuestas "Si" entonces *aceptamos* el valor de p . Si repetimos este proceso miles de veces con valores de p aleatorios todo los valores aceptados de p nos terminan dando la distribución de la probabilidad de que a la gente le gusta Arjona:

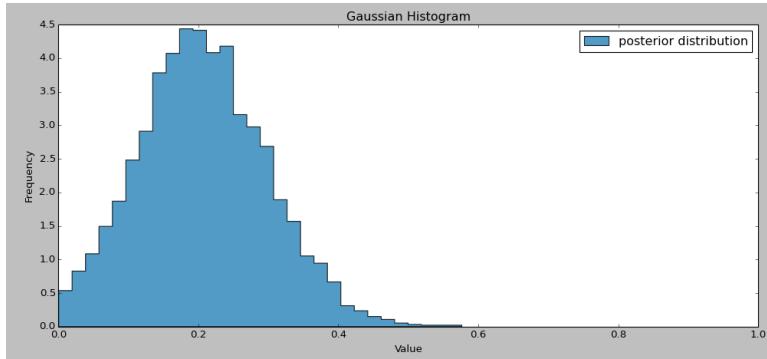


Figure 1.31: Distribución a-priori

La figura 1.31 nos muestra el resultado de la simulación. Como podemos ver la misma está centrada en 0.2 que es el valor que habíamos calculado analíticamente pero ahora podemos ver también qué grado de incertidumbre tenemos. Podemos ver que casi con certeza el valor de p no puede ser 0.6 o mayor y que la mayoría de los valores están entre 0.05 y 0.4. Esta información es extremadamente valiosa ya que nos permite experimentar con otros espacios muestrales y algoritmos.

Lo que MCMC hace es simplemente "navegar" aleatoriamente por diferentes valores que puede tomar nuestra distribución a-priori " p " y verificar si esos valores son buenos o malos de acuerdo a la distribución a-posteriori (en nuestro caso 35/100).

Pensemos ahora el caso en el cuál tenemos no una sino varias distribuciones a-priori que queremos averiguar, el algoritmo empieza a tener problemas porque la cantidad de valores a probar crece de forma exponencial y es probable que incluso luego de millones de simulaciones no podamos converger a un valor adecuado para nuestras distribuciones. Es aquí donde aparece la verdadera inteligencia del algoritmo MCMC que no es ni mas ni menos que aplicar la heurística Metrópolis-Hastings que ya hemos visto.

El algoritmo va a empezar con ciertos valores al azar para las distribuciones desconocidas, que pueden tomarse de cualquier distribución, en nuestro caso la distribución es uniforme porque todos los valores de p son equiprobables pero podría ser una Poisson, Exponencial, etc. Una vez que el algoritmo tiene un estado "inicial" puede calcular la probabilidad a-posteriori de dicho estado. Con este dato el algoritmo ahora sugiere un cambio, es decir una variación en una de las probabilidades y calcula la nueva probabilidad a-posteriori, si es mejor a la anterior entonces el algoritmo acepta el cambio, si es peor entonces lo acepta con una probabilidad que es p_i/p_{i-1} donde sabemos que $p_{i-1} > p_i$ ya que sino hubiesemos aceptado el cambio automáticamente.

Siguiendo este proceso MCMC va convergiendo a los valores de las n distribuciones a-priori que son mas probables y nos devuelve, igual que en nuestro

caso los distintos "samples" de estas probabilidades.

Supongamos que tenemos dos distribuciones a-priori ambas uniformes. Nuestro "espacio" es un cuadrado en dos dimensiones en donde cualquier punto del cuadrado es un par de probabilidades válido. Para cada punto de este cuadrado hay un cierto valor a-posteriori que podemos pensar como la elevación de una montaña sobre el cuadrado. El objetivo de MCMC es encontrar la montaña, por eso va navegando el cuadrado intentando subir, para encontrar los valores de p_1 y p_2 que maximicen la distribución a-posteriori.

MCMC es un algoritmo extremadamente poderoso ya que nos permite a partir de los resultados observados de un cierto experimento obtener cuáles fueron las probabilidades a-priori que generaron dichos resultados con mayor probabilidad. Esto puede usarse en una gran cantidad de aplicaciones y es el algoritmo mas importante para realizar análisis de datos bayesianos.

Habiendo explicado la intuición del algoritmo debemos aclarar que es muy complejo programar MCMC de forma eficiente por lo que recomendamos usar alguna biblioteca o función que esté bien testeada y optimizada. PyMC en Python es un buen ejemplo.

1.8 Gibbs Sampling

Una forma de aplicar MCMC es simulando valores de nuestro set de datos, valores nuevos no aquellos que ya conocemos. Para hacer esto tenemos que conocer la distribución de probabilidades del set de datos y no la tenemos. Los casos posibles son demasiados ya que cada punto es una variable de n dimensiones en donde cada dimensión puede tomar muchos valores posibles por lo que sería imposible generar todos los casos posibles y luego tomar uno al azar de estos.

La solución para el problema de generar muestras suele ser emplear el método conocido como *Gibbs sampling*, en este método vamos a elegir un dato al azar de nuestro set de datos como punto de partida, luego vamos a elegir una variable al azar (columna) y para esa variable vamos a generar un valor nuevo. El valor nuevo lo generamos en base a la distribución de probabilidades fijando las otras variables del punto seleccionado.

1.9 La Unión hace la fuerza: Ensamblés

La última sección de nuestro largo capítulo introductorio la dedicamos a explicar brevemente un concepto que es fundamental en todo proceso de Data-Science: el resultado de varios algoritmos combinados casi siempre es mejor que el resultado de cualquier algoritmo individualmente.

Este concepto se ha demostrado empíricamente en muchísimas ocasiones, en el famoso concurso del premio Netflix que otorgó un millón de dólares a los ganadores el equipo ganador usó un ensamble de mas de 200 algoritmos diferentes. El mejor algoritmo de compresión hasta la fecha (PAQ) utiliza un

ensamble de varios algoritmos diferentes para poder comprimir cada bit del archivo.

El principio detrás del éxito de los ensambles es muy simple: cada algoritmo explota particularmente algunas propiedades de nuestro set de datos, es decir que para diferentes datos diferentes algoritmos funcionan mejor, como lo que tenemos es un conjunto de datos la combinación de varios algoritmos suele ser la que nos de los mejores resultados.

Mas adelante veremos diferentes métodos para crear ensambles de algoritmos y de esta forma sacar provecho de la unión de varios modelos diferentes para una solución común.

Como cierre de este capítulo que ha sido tan largo y tan extenso queremos quedarnos con los dos conceptos mas importantes que hemos visto:

1. Siempre es bueno conseguir mas datos a punto tal que conseguir mas datos suele ser mejor que conseguir un mejor algoritmo
2. La unión de varios algoritmos suele ser mejor que el resultado individual de cada uno de ellos.

Chapter 2

Nuestro Primer Algoritmo:KNN

En este capítulo vamos a presentar el primero de los muchos algoritmos que vamos a estudiar a lo largo del curso, elegimos este algoritmo por ser muy simple de entender y porque suele dar muy buenos resultados aun en problemas complejos, es un algoritmo que permite realizar tareas tanto de clasificación como de regresión y que también es usado luego en otros algoritmos como parte del proceso. Mencionamos todo esto para destacar la importancia que tiene y la necesidad de conocerlo en detalle.

El algoritmo se denomina KNN (K-Nearest-Neighbors) ya que se basa en encontrar para un determinado punto sus K -vecinos mas cercanos. Asumimos que nuestro set de datos está formado por un conjunto de m puntos en n dimensiones siendo todos los valores numéricos.

Para poder usar KNN hay que definir dos cosas: la métrica a usar para calcular las distancias y el valor de k es decir cuantos vecinos vamos a considerar. A modo de ejemplo veamos un caso en el cual usamos KNN para clasificación usando la distancia euclidiana.

K	P(Blue)	P(Red)
1	0	1
3	1/3	2/3
5	3/5	2/5

Table 2.1: Ejemplo KNN.

La tabla nos muestra para diferentes valores de k cual es la probabilidad de cada una de las clases, en nuestro caso rojo o azul. Como podemos ver para el punto verde si $k = 1$ tomamos únicamente el punto mas cercano y determinamos que la clase es la de dicho punto, en nuestro caso rojo con probabilidad 1 y por lo tanto la probabilidad de azul es cero. Si tomamos $k = 3$ entonces tomamos los

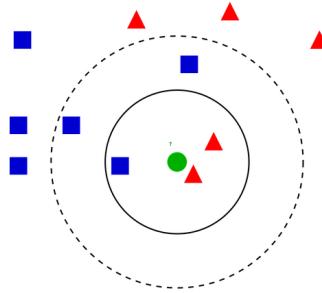


Figure 2.1: KNN.

tres vecinos mas cercanos, dos son rojos y uno es azul por lo tanto la probabilidad de rojo es $2/3$ y la probabilidad de azul es $1/3$. Finalmente con $k = 5$ tenemos tres puntos azules y dos rojos por lo que ahora la clase mas probable es azul con probabilidad $3/5$ y rojo tiene probabilidad $2/5$.

Cuando queremos simplemente clasificar un punto cuya clase no conocemos no hacen falta las probabilidades podemos simplemente asignarlo a la clase con mayoría entre los k -vecinos del punto.

Si nuestro problema es de regresión entonces podemos predecir para nuestro punto el promedio de los valores de los k -vecinos mas cercanos.

Con este sencillo esquema tenemos un algoritmo muy útil tanto para problemas de clasificación como de regresión, pero KNN no solo se usa como algoritmo de clasificación y regresión sino que es también parte fundamental de otros algoritmos. Por ejemplo podemos usar KNN en el marco de un sistema de recomendaciones, tomando las k películas mas similares a las que le gustan al usuario para recomendarlas o tomando los k usuarios mas parecidos al usuario para recomendarle películas que estos hayan visto. Es por este motivo que KNN es un algoritmo sumamente importante y que estudiaremos en detalle.

2.1 La Métrica a emplear

La métrica a emplear para las distancias en KNN es sumamente importante, puede usarse cualquier métrica que cumpla las siguientes propiedades:

1. $d(x, y) \geq 0$
2. $d(x, y) = d(y, x)$
3. $d(x, y) \leq d(x, z) + d(z, y)$

Las tres propiedades son conocidas, una distancia debe ser positiva, simétrica y debe cumplir la desigualdad triangular. Dado que vamos a usar distancias no solo en KNN sino en muchos algoritmos a lo largo del curso es conveniente

hacer un listado de las distancias mas populares y sus definiciones para poder mencionar en que casos es posible usarlas.

2.1.1 Distancia Minkowsky

La distancia mas general es la llamada distancia de Minkowsky cuya definición es la siguiente:

$$\left(\sum_i^n |x_i - y_i|^p \right)^{\frac{1}{p}}$$

Cuando $p = 2$ la distancia Minkowsky es la conocida distancia euclídea que es sin dudas la mas utilizada. Hay otros tres casos interesantes de esta distancia:

2.1.2 Distancia Manhattan

En esta distancia $p = 1$, la distancia Manhattan se llama así porque es la forma de calcular distancias en una ciudad con forma de grilla en la cual solo nos podemos mover por las calles en forma horizontal y vertical, la distancia entre dos puntos es entonces la sumatoria de la diferencia entre todas sus coordenadas.

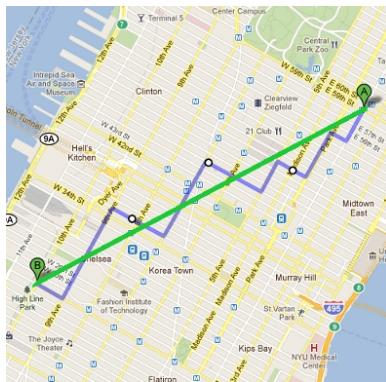


Figure 2.2: Distancia Manhattan vs Distancia Euclideana.

2.1.3 Distancia de Hamming o Norma l_0

Esta es la distancia de Minkowsky cuando $p = 0$, en este caso la distancia entre dos puntos es igual a la cantidad de elementos en las cuales difieren los vectores. La distancia de Hamming en general se define y usa para vectores binarios, formados por ceros y unos pero la definición puede extenderse a vectores con valores arbitrarios contando simplemente en cuantas dimensiones los valores de los vectores son diferentes.

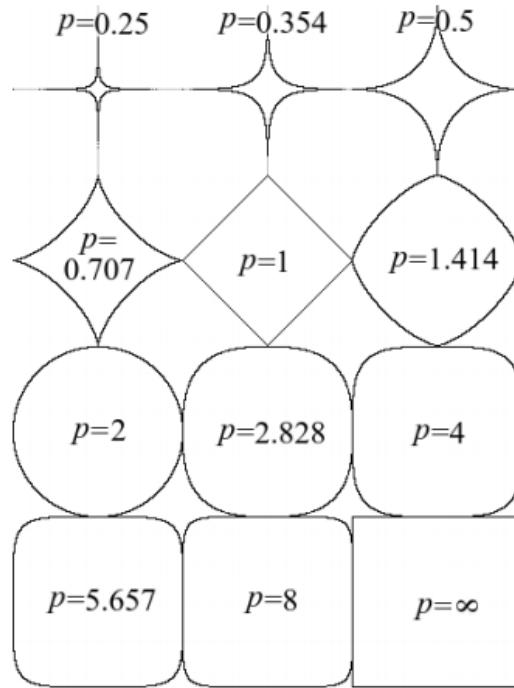


Figure 2.3: Puntos a distancia=1 del origen para diferentes valores de p en la distancia Minkowsky, observar que la distancia euclídea ($p = 2$) es invariante a la rotación lo cual la hace tan popular

2.1.4 Norma l_∞

En este caso $p = \infty$ y la distancia equivale a la diferencia más grande entre dos elementos cualesquiera entre los vectores. Esto es fácil de ver ya que a medida que aumentamos p en la distancia Minkowsky solo las coordenadas con mayor diferencia tienen peso en la distancia final, en el límite la única coordenada que realmente cuenta es la de mayor distancia.

Sean por ejemplo los siguientes vectores:

$$\mathbf{X} = (1, 0, 2, 4)$$

$$\mathbf{Y} = (1, 2, 1, 1)$$

La distancia de Hamming entre los vectores es 3 porque difieren en tres de las 4 dimensiones. La distancia Manhattan es 6 que es la sumatoria de las distancias entre cada dimensión, la distancia euclídea es $\sqrt{14}$ y finalmente la norma l_∞ es 3 que es la mayor diferencia entre las coordenadas de los vectores.

2.1.5 Distancia de Mahalanobis

La distancia de Mahalanobis es útil cuando tenemos atributos (features) que están correlacionados. Consideremos por ejemplo el gráfico de la figura 2.4. Como podemos ver hay una correlación entre los ejes X e Y, los dos puntos marcados con las X rojas están dentro del mismo percentil de variabilidad de cada uno de sus ejes. En otras palabras en una escala relativa a la variabilidad de cada eje los dos puntos deberían estar a igual distancia del origen.

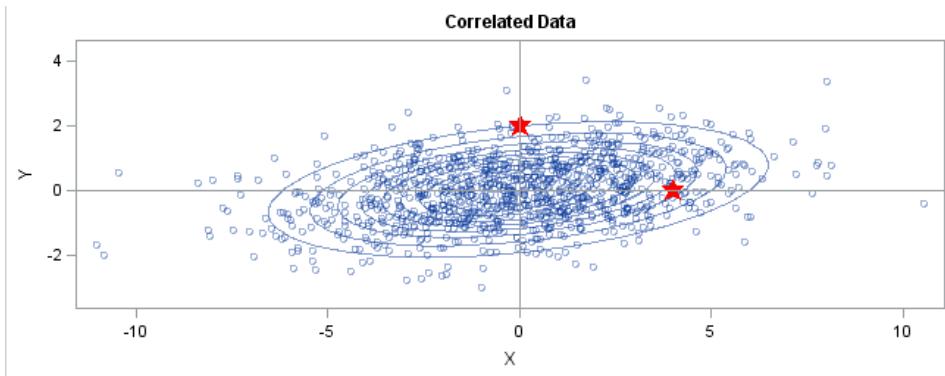


Figure 2.4: Distancia de Mahalanobis.

La distancia de Mahalanobis es aquella que teniendo en cuenta la variabilidad de cada uno de los atributos logra que puntos que están dentro del mismo percentil de variación para su atributo queden a la misma distancia. La fórmula para la distancia de Mahalanobis es:

$$D(x, y) = \sqrt{(x - y)S^{-1}(x - y)}$$

Donde "S" es la matriz de covarianza. La matriz de covarianza se define como aquella matriz que tiene en la diagonal la varianza de cada uno de los atributos y en los elementos A_{ij} la covarianza entre los atributos "i" y "j". La matriz de covarianza se define de la siguiente forma:

$$\text{cov}(x, y) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

Hay que destacar que cuando la matriz de covarianza es la identidad la distancia de Mahalanobis es igual a la distancia euclídea.

2.1.6 Distancia Coseno

La distancia coseno mide la distancia entre dos vectores como el ángulo entre los mismos. Recordemos que el coseno entre dos vectores puede calcularse de la siguiente forma:

$$\cos \theta = \frac{\langle X, Y \rangle}{\|X\| \|Y\|}$$

Es decir el producto interno entre los vectores dividido el producto de sus normas. La distancia coseno es entonces:

$$\theta = \arccos \frac{\langle X, Y \rangle}{\|X\| \|Y\|}$$

Notemos que el coseno es una medida de semejanza (a mayor coseno mas chico el ángulo entre los vectores) mientras que el ángulo es una medida de distancia. En muchos algoritmos vamos a usar esta dualidad entre semejanzas y distancias calculando unas u otras según sea necesario.

Observemos también que si los vectores están normalizados el coseno entre los mismos puede calcularse simplemente como el producto interno entre los vectores, esta propiedad también será muy usada a lo largo del curso.

La distancia coseno se usa cuando lo que importa es la dirección en la que apuntan los vectores y no la magnitud de los mismos. Es una distancia muy usada en textos, por ejemplo en el modelo Bag of Words ya que lo que nos importa es la diferencia semántica entre los textos y no si un texto es mas largo que el otro. Es decir que la cantidad de palabras que el texto tiene no debería influir en el cálculo de distancias.

La distancia coseno está relacionada con el coeficiente de correlación de Pearson que se define de la siguiente forma:

$$\rho(x, y) = \frac{\sum_i (x_i - \mu_x)(y_i - \mu_y)}{\sqrt{\sum_i (x_i - \mu_x)^2} \sqrt{\sum_i (y_i - \mu_y)^2}} \quad (2.1)$$

μ_x y μ_y son los promedios de los valores del vector. Si los promedios son iguales a cero entonces el coeficiente de Pearson es igual al coseno entre los vectores. El coeficiente de Pearson toma valor entre -1 y 1 y es una medida de semejanza.

La distancia coseno o el coeficiente de Pearson son especialmente útiles cuando nuestros vectores tienen muchos elementos desconocidos, o lo que es igual, son muy dispersos, con mayoría de ceros. En estos casos si calculamos la distancia euclídea solo para los elementos del vector que son conocidos los vectores que tienen muy pocos elementos (o muy pocos distintos de cero) van a estar muy cerca de todos los demás. Es por esto que el coseno o Pearson son usados en sistemas de recomendaciones como veremos mas adelante en el texto.

2.1.7 Distancia de Edición o Levenshtein

La distancia Levenshtein es una forma de calcular la distancia entre strings. Lo que medimos es la cantidad de operaciones que hay que hacer para convertir un string en otro. Las operaciones válidas son agregar un carácter al string o quitar un carácter del string, ambas tienen costo 1. La distancia total es la suma de los costos, es decir el total de inserciones o eliminaciones a realizar. En

algunos casos puede considerarse también el reemplazar un carácter por otro, esta operación puede considerarse de costo igual a 1 o bien de costo igual a 2 razonando que una modificación es igual a una eliminación seguida de una inserción. Ambas definiciones son válidas y puede usarse indistintamente.

Como ejemplo la distancia Levenshtein entre "pato" y "palta" es 3, ya que necesitamos tres operaciones de costo 1 cada una para pasar de un string al otro "pato", "palto", "palt", "palta".

La distancia Levenshtein tiene la particularidad de que es costoso calcularla, en general por programación dinámica y no existen aproximaciones eficientes a la misma. Es una distancia muy útil pero que debe ser usada con cuidado cuando los datos son masivos debido al costo computacional de la misma.

2.1.8 Distancia de Jaccard

La distancia de Jaccard se usa para calcular la distancia entre conjuntos. La definición es muy simple:

$$dJ(X, Y) = 1 - \frac{X \cap Y}{X \cup Y}$$

En un abuso de notación debemos entender que el numerador indica la cantidad de elementos en la intersección y el denominador la cantidad de elementos en la unión de los conjuntos. La semejanza de Jaccard es simplemente $1 - dJ(X, Y)$ y es un número entre 0 y 1 y lo denominamos $J(X, Y)$. Cuando los conjuntos son idénticos la cardinalidad de la intersección es igual a la cardinalidad de la unión y la distancia es 0 y la semejanza 1. Cuando los conjuntos son disjuntos la distancia es 1 y la semejanza 0.

Ejemplo: $X=1,3,4,7$ $Y=1,2,3,5$ $dJ(X, Y) = 4/6$, hay dos elementos en la intersección de los conjuntos (1,3) y 6 elementos en la unión (1,2,3,4,5,7) por lo tanto la semejanza es $2/6$ y la distancia $4/6$.

2.1.9 Distancia Geodésica

La distancia geodésica se aplica en grafos y es simplemente la cantidad de aristas a recorrer para llegar desde un nodo a otro. Esta distancia se usa cuando no tenemos información sobre las coordenadas de los nodos sino simplemente la información de que nodos están vinculados con otros.

En la figura 2.5 la distancia geodésica entre "e" y "c" es 3 ya que hay que recorrer tres aristas para llegar desde uno nodo al otro.

2.1.10 Distancia entre Grafos

Esta distancia sirve para comparar un grafo con otro, es decir que queremos saber que tan diferentes son dos grafos cualesquiera. La definición es similar a la distancia de edición entre strings, contamos el costo de las operaciones necesarias para convertir un grafo en otro. Las operaciones posibles son agregar

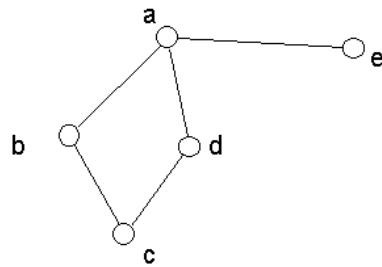


Figure 2.5: Distancia Geodésica.

un nuevo nodo, eliminar un nodo, agregar una arista o eliminar una arista, todas ellas de costo 1.

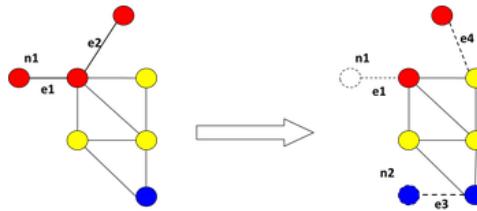


Figure 2.6: Distancia entre grafos.

En la figura 2.6 vemos dos grafos a distancia 6, ya que para pasar del primero al segundo son necesarias las siguientes operaciones:

1. eliminar la arista e1
2. eliminar el nodo n1
3. eliminar la arista e2
4. agregar la arista e4
5. insertar el nodo n2
6. insertar la arista e3

2.1.11 Distancia para atributos categóricos: VDM

Hasta el momento no hemos visto ninguna distancia que tenga sentido para atributos categóricos o factores es decir aquellos que pueden tomar un valor

de una lista predeterminada de valores. Temas como cuál es la distancia entre "masculino" y "femenino" o entre "rojo" y "verde" no son conceptos para nada claros. Para este tipo de atributos vamos a definir la distancia VDM (value distance measure) de la siguiente forma:

$$VDM(f1, f2) = \sum_{class=i}^c |P(f1|class_i) - P(f2|class_i)|$$

Consideremos el siguiente ejemplo en el cual tenemos dos clases: "azul" y "rojo" y un atributo categórico con tres valores posibles: "A", "B" o "C". Supongamos los siguientes puntos:

(rojo, A)
 (azul, B)
 (rojo, C)
 (azul, B)
 (azul, C)

Podemos calcular la probabilidad de cada uno de los valores posibles para el atributo en función de cada clase, por ejemplo $P(A|rojo)$ es la probabilidad del atributo "A" si la clase es rojo. Para la clase rojo nuestro atributo toma una vez el valor "A" y una vez el valor "C" por lo tanto $P(A|rojo) = 1/2$. La tabla de probabilidades es:

	rojo	azul
A	1/2	0
B	0	2/3
C	1/2	1/3

Table 2.2: Probabilidades de factores.

A partir de esta tabla podemos calcular las distancias A-B, A-C y B-C usando la fórmula de la VDM. Por ejemplo para la distancia entre A y B tenemos que hacer la sumatoria de la diferencia entre la probabilidad de "A" y "B" en cada una de las clases es decir: $VDM(A, B) = |(1/2 - 0)| + |(0 - 2/3)| = 1/2 + 2/3 = 7/6$. La tabla de distancias es la siguiente:

	A	B	C
A	0	7/6	1/3
B	7/6	0	5/6
C	1/3	5/6	0

Table 2.3: Ejemplo distancia VDM.

Es decir que "A" está mas cerca de "C" que de "B"

2.1.12 Distancias Especiales

En muchos casos es necesario definir una distancia mixta o customizada de acuerdo a la aplicación que tenga nuestro set de datos. Consideremos por ejemplo un set de datos sobre películas, para cada película conocemos su director, género, actores principales, actores de reparto, estudio, año de publicación, guionista, sonidista, etc etc. Es decir que tenemos una buena cantidad de atributos de tipos realmente muy diferentes. La dificultad pasa por calcular la distancia entre dos películas o lo que es lo mismo la semejanza entre dos películas.

Este tipo de problemas necesitan de la definición de una distancia "custom" o definida por el usuario. Por ejemplo podemos asignar "pesos" a cada uno de los diferentes atributos y posibles valores de semejanza. Si las películas tienen el mismo director entonces 5 puntos, sino 0 puntos. Si no tienen actores en común 0 puntos, 1 punto si tienen un actor en común y 2 puntos si tienen 2 o mas actores en común. Si el año es el mismo 3 puntos, si la diferencia es de 1 o 2 años 2 entre 3 y 5 años 1 punto y mas de 5 años 0 puntos, etc. De esta forma definimos la semejanza máxima que puede haber entre dos películas como el máximo de puntos que podemos sumar cuando todos los atributos son idénticos. Luego la semejanza entre dos películas se puede calcular como la suma de puntos sobre el total posible y la distancia es 1-semejanza ya que la semejanza es un número entre 0 y 1. Esta construcción de distancias o semejanzas "custom" es clave en sistemas de recomendación basados en contenidos en donde queremos recomendarle a un usuario productos similares a aquellos que le han gustado o ha comprado recientemente. Este tema será tratado con mayor extensión en el capítulo sobre sistemas de recomendación.

2.1.13 Aprendiendo la Distancia

Una posibilidad intrigante para la distancia entre los puntos es aprenderla a partir de los puntos mismos. Dados los puntos y la clase a la que pertenece cada uno podemos usar un algoritmo de clasificación conocido como "Random Forests" que será desarrollado mas adelante para aprender un modelo a partir de nuestros datos. En un Random Forest obtenemos un conjunto de n árboles de decisión en donde cada árbol intenta clasificar a nuestros puntos. Tomando dos puntos podemos calcular su distancia como el número de árboles en los que RF predice una clase diferente para cada punto. Normalizando sobre la cantidad de árboles n tenemos una distancia entre 0 y 1.

Este tipo de distancias que aprendemos con un algoritmo para aplicar luego otro algoritmo deben ser tomadas con cuidado, por ejemplo no tendría sentido aprender una distancia con RF para luego aplicar KNN para clasificar si el resultado del Random Forest es superior al que obtenemos con KNN.

2.2 Determinando la Distancia a usar en KNN

Tenemos que comentar ahora de que forma determinar la distancia a usar en KNN, el método básico es probando diferentes distancias y viendo cual de ellas nos da un mejor resultado. Dada una cierta distancia de un cierto valor de k podemos ver para cada punto de nuestros datos de entrenamiento, para los cuales conocemos su clase (o valor si es un problema de regresión) cual sería la predicción del algoritmo KNN. Luego simplemente comparamos las predicciones con los valores conocidos para calcular la precisión del algoritmo o el error del mismo. En un problema de clasificación podemos calcular el porcentaje de puntos que el algoritmo clasifica correctamente mientras que en un problema de regresión podemos calcular la sumatoria total de los errores al cuadrado sobre el total de puntos (MSE o Mean Squared Error).

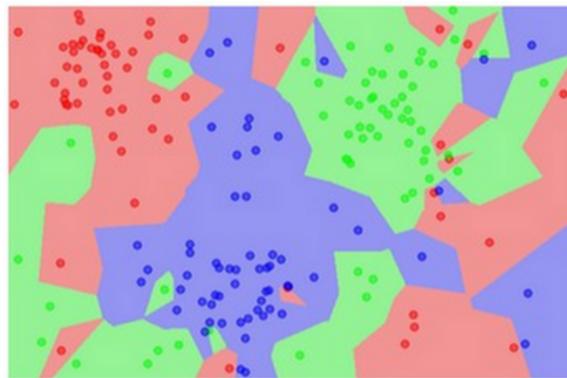
Notemos que la precisión de KNN puede calcularse directamente a partir del set de entrenamiento, en el set de entrenamiento cada punto tiene una clase, lo que hacemos es tomar cada punto y clasificarlo en base a sus k vecinos mas cercanos y comparar esta predicción con la clase real del punto que conocemos. Este mecanismo nos permite analizar el comportamiento de KNN en el set de entrenamiento y nos ayuda a elegir los mejores hiper-parámetros para el algoritmo. Al usar este método de validación cuando tomamos cada punto no lo consideramos como parte de sus k -vecinos, es decir que hacemos de cuenta que el punto está fuera del set de entrenamiento. Si los puntos fueran muchos podemos seleccionar algunos al azar para calcular la precisión del algoritmo (Montecarlo style)

2.3 Determinar el valor de K

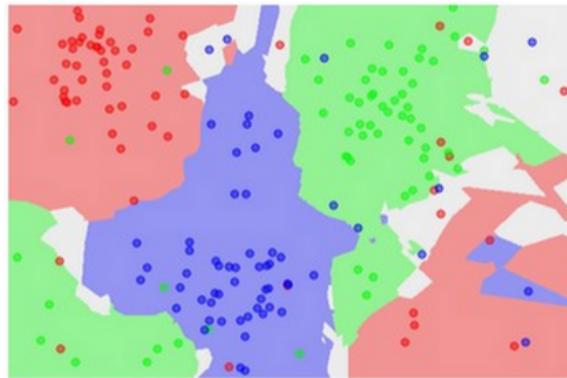
Para determinar el valor óptimo para K el procedimiento es igual que para determinar la métrica a usar como distancia: probar diferentes valores de k y ver cual es el que nos da mejores resultados. Aquí hay que tener un cierto cuidado ya que entender qué implica aumentar o disminuir la cantidad de vecinos mas cercanos y es la primera vez que encontramos el concepto de sobre-ajuste (*overfitting*) al cual nos referiremos de aquí en mas como overfitting por ser el término mas comúnmente usado en la literatura para describir el problema.

Veamos qué pasa cuando usamos $k = 1$ es decir cuando a cada punto lo clasificamos únicamente en base al punto mas cercano.

Lo que podemos ver es que la distribución que aprende nuestro clasificador no es homogénea, es decir que hay puntos de distintas clases mezclados en zonas en las cuales predomina otra clase, esto implica que nuestro clasificador va a funcionar bien o muy bien para el set de entrenamiento pero no ha aprendido a generalizar y esto quiere decir que no va a ser muy bueno para predecir la clase de puntos nuevos que no hayamos observado en el set de entrenamiento. Esta es la definición de *overfitting*: cuando un algoritmo funciona bien o muy bien para el set de entrenamiento y mal para datos nuevos. El concepto es que aprender a predecir el set de entrenamiento no es lo mismo que aprender a generalizar. En

Figure 2.7: KNN con $k = 1$.

KNN para generalizar mejor hay que usar valores de k mayores. Veamos qué pasa con $k = 5$

Figure 2.8: KNN con $k = 5$.

Como podemos ver en la figura 2.8 el clasificador ha aprendido ahora a generar áreas más suaves y por consiguiente es un clasificador que generalizará mejor para predecir puntos que no estaban en el set de entrenamiento. Podríamos pensar que es entonces conveniente usar valores de k grandes como $k = 1000$ o incluso $k = n$ sin embargo las cosas no son tan sencillas. Al aumentar el valor de k estamos dando cada vez mayor peso a las clases que tienen mayor cantidad de puntos en el set de entrenamiento, en el extremo si $k = n$ vamos a predecir para todos los puntos la clase que mayor cantidad de puntos tiene en el set de entrenamiento lo cual no es bueno. En definitiva el k óptimo en KNN es aquel que nos de un buen desempeño en cuanto a la precisión de

clasificación para el mayor k posible. Una forma de encontrar este valor óptimo es seleccionar el mayor k que nos de una precisión aceptable para el problema que estemos trabajando.

Veamos un ejemplo de la precisión de un clasificador de tipo KNN para diferentes valores de k

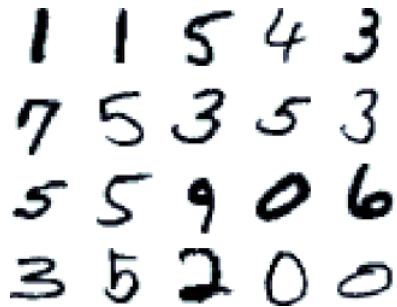


Figure 2.9: MNIST dataset

En este caso usamos KNN para predecir el valor de dígitos manuscritos en el set de datos MNIST. El set de entrenamiento cuenta con 20.000 imágenes de 28x28 pixeles cada una representando dígitos manuscritos. Cada imagen tiene como clase el dígito que representa (0 a 9). El set de test tiene otras 10.000 imágenes cuya clase no conocemos. Aplicamos KNN para determinar a qué clase corresponde cada dígito en base a la clase de sus k vecinos mas cercanos usando la distancia euclíadiana.

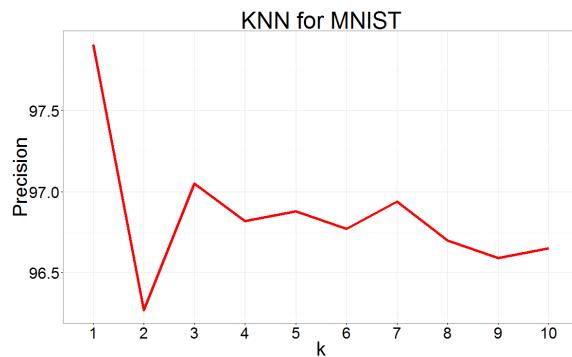


Figure 2.10: KNN para MNIST

Observamos en primer lugar que KNN tiene un resultado muy bueno para este set de datos, esto se debe a que el set de datos está presentado de forma ideal para KNN mas adelante explicaremos el motivo. En cuanto al valor de k si $k = 1$ tenemos un caso de overfitting, $k = 2$ no funciona nada bien y luego

podríamos llegar a elegir prácticamente cualquier valor entre $k = 3$ y $k = 11$ ya que todos esos valores tienen un muy buen nivel de precisión y también generalizan bien.

2.3.1 Grid Search y Random Search en KNN

En un algoritmo vamos a llamar parámetros a aquellos valores que el algoritmo encuentra a partir de los datos y vamos a llamar hiper-parámetros a aquellos datos que el algoritmo necesita para poder funcionar. KNN tiene dos hiper-parámetros: k y la métrica a usar (distancia). Para encontrar los hiper-parámetros óptimos para un algoritmo pueden usarse dos métodos: Grid-Search o Random-Search. En un Grid-Search probamos todas las combinaciones posibles dentro de una lista de valores posibles para cada hiper parámetro. Si por ejemplo probamos cuatro métricas y diez valores de k , tenemos que correr KNN un total de 40 veces. Es posible empezar con un "paso" grueso para los hiper parámetros y luego ir refinando la búsqueda, por ejemplo podemos probar primero con $k = 1, 5, 10, 15, 20$ y luego si el óptimo resultó ser por ejemplo 10 buscar entre $k = 6$ a $k = 14$. Este proceso es especialmente necesario cuando los hiper-parámetros pueden tomar valores reales.

Algorithm 7: Grid Search for KNN

Data: X: training set, Y: labels,m:number of observations
Result: k and distance metric for KNN

```

1 best_precision = 0, best_k = 0,best_metric = none;
2 for k in (1..100) do
3   for metric in (euclidean,manhattan, cosine, jaccard,hamming,linf,...)
4     do
5       preds = knn(X,Y,k,metric);
6       precision = sum(preds==Y)/m;
7       if precision > best_precision then
8         best_precision = precision;
9         best_k = k;
10        best_metric = metric ;

```

Cuando la cantidad de hiper-parámetros es realmente muy grande la combinatoria a realizar puede resultar muy ineficiente, en estos casos puede recurrirse al método de Random-Search. En un Random-Search controlamos cuantas iteraciones realizamos de nuestro algoritmo y por cada iteración seleccionamos los valores de los hiper-parámetros al azar dentro de un rango pre establecido. Este método no es tan preciso como un grid-search pero es mucho mas rápido.Random-Search es nuestro primer ejemplo de algoritmo aleatorizado una rama de los algoritmos que usaremos extensamente en el curso.

2.4 Sensibilidad a valores fuera de escala

Es hora de hablar de los puntos débiles de KNN y qué tipo de cosas pueden hacer que el algoritmo tenga malos resultados. Un primer factor importante es que dado que KNN se basa en distancias para funcionar correctamente, queremos que todos los atributos de nuestros datos tengan el mismo peso en el cálculo de distancias, si los atributos no están todos dentro de la misma escala entonces puede que un atributo domine el cálculo de distancias sobre todos los demás y el algoritmo tendrá un rendimiento pobre.

Para evitar que un atributo domine el cálculo de distancias es necesario normalizar los valores de los atributos (columnas), por ejemplo al rango [0..1]. Esto se logra restándole a cada atributo el promedio de la columna y luego dividiendo por la desviación standard de la misma. Al restar el promedio logramos que el promedio de la columna sea cero y al dividir por la desviación standard logramos que el rango sea [0..1]. A este procedimiento lo llamaremos "Normalización de atributos o features" o simplemente "Normalización". Son muchos los algoritmos que necesitan datos normalizados para funcionar mejor. Notemos que normalizar no cambia las propiedades geométricas de los datos es decir que las distancias se mantienen.

2.5 Sensibilidad a atributos anómalos

Mas allá de haber normalizado los datos es posible que no todos los atributos sean adecuados para clasificar. Consideremos por ejemplo un caso muy simple en el cual estamos intentando predecir la altura de alumnos en base a otros datos como el peso, edad, sexo, etc. Atributos del estilo "educación" o el promedio en la carrera o las notas que tenga claramente no son relevantes para la tarea que estamos desarrollando. En muchos casos esto no es evidente ya que desconocemos cuáles son los atributos mas relevantes y cuáles son no-relevantes de acuerdo a lo que queremos predecir. Los algoritmos de Forward Selection y Backward Selection permiten determinar cuál es el conjunto de atributos óptimo para KNN.

2.5.1 Forward Selection

En el algoritmo de forward selection comenzamos con cero atributos y en cada paso agregamos el atributo que mejor resultado nos genera. Para considerar el resultado podemos usar un valor fijo para k o bien la mejor precisión para varios valores de k o bien un promedio de la precisión para diferentes valores de k probados. De esta forma vamos iterando y agregando atributos siempre y cuando los resultados mejoren, cuando ya no es posible mejorar los resultados o bien cuando no quedan mas atributos por agregar termina el algoritmo y tenemos el set de atributos a usar en KNN.

Algorithm 8: Forward Selection

Data: data:X, labels:Y, number of observations:m
Result: Set of features to use

```

1 unused_features=[all features], used_features=[], best_precision = 0;
2 while not empty unused_features do
3     loop_precision = 0;
4     best_at = NULL;
5     for at in unused_features do
6         pred = knn(X[used_features + at],Y,k,metric);
7         precision = sum(pred==Y)/m;
8         if precision > loop_precision then
9             loop_precision = precision;
10            best_at = at;
11        if loop_precision > best_precision then
12            add best_at to used_features;
13            remove best_at from unused_features;
14            best_precision = loop_precision;
15        else
16            EXIT #No improvement adding any feature

```

2.5.2 Backward Selection

Backward selection es la versión inversa del algoritmo forward selection. Comenzamos con todos los features y vamos eliminando un feature a la vez hasta que ya no se pueda mejorar la precisión del algoritmo.

2.6 Aproximaciones para KNN

2.6.1 Orden del algoritmo y eficiencia

Es necesario ahora analizar la eficiencia del algoritmo KNN, en su versión mas simple para clasificar un punto nuevo tenemos que compararlo contra todos los puntos existentes (m) para encontrar los k vecinos mas cercanos. Cada una de estas m comparaciones implica comparar n dimensiones. Por lo tanto el orden del algoritmo es $O(m * n)$

La necesidad de comparar cada punto a clasificar contra todos los puntos del set de datos es el gran problema de KNN y el motivo por el cual suele ser muy ineficiente para datos masivos. Clasificar 1000 puntos con un set de entrenamiento de un millón de datos implicaría un total de mil millones de comparaciones cada una de ellas en n dimensiones. Esto nos lleva a la conclusión de que KNN no escala bien y que a partir de una cierta cantidad de datos es necesario usar algún tipo de optimización o aproximación que nos permita lograr mejores resultados. Veremos a continuación algunas de estas posibles

Algorithm 9: Backward Selection

Data: data:X, labels:Y, number of observations:m
Result: Set of features to use

```

1 used_features=[all_features], best_precision = 0;
2 while True do
3     loop_precision = 0;
4     best_at = NULL;
5     for at in used_features do
6         pred = knn(X[used_features - at],Y,k,metric);
7         precision = sum(pred==Y)/m;
8         if precision > loop_precision then
9             loop_precision = precision;
10            best_at = at;
11        if loop_precision > best_precision then
12            remove best_at from used_features;
13            best_precision = loop_precision;
14        else
15            EXIT #No improvement removing any feature

```

optimizaciones y aproximaciones.

2.6.2 Indices Espaciales: KD-Trees

Un índice espacial es una estructura de datos en donde podemos insertar nuestros puntos n-dimensionales de forma tal de luego poder realizar búsquedas de los puntos mas cercanos a un cierto query sin tener que recorrer todos los puntos del set de datos. Existen muchísimas variantes de estos índices, a efectos de ejemplo vamos a mencionar el mas popular que son los llamados KD-Trees. Por motivos que vamos a aclarar muy pronto vamos a ser muy breves en cuanto a la descripción del funcionamiento de un KD-Tree.

Un KD-Tree es similar a un árbol binario de búsqueda pero en cada nivel del árbol comparamos una coordenada diferente de nuestros puntos n-dimensionales. Si trabajamos en dos dimensiones la raíz del árbol compara la coordenada "X" del punto, todos los puntos con X menor al valor de la raíz van a la rama izquierda y todos los puntos con X mayor o igual al valor de la raíz van a la rama derecha. En el siguiente nivel comparamos la coordenada Y. Todos los puntos con Y menor al valor del nodo van a la izquierda y los mayores o iguales a la derecha. En el siguiente nivel volvemos a la coordenada "X" y así sucesivamente hasta que cada rama tenga un solo punto o bien hasta un máximo número de niveles prefijados.

Conociendo los puntos a insertar podemos tomar como valor de "split" en cada nivel al punto para el cual la coordenada con la cual estamos trabajando es la media. La mediana funciona mejor que el promedio porque nos permite

siempre dividir el conjunto de puntos en dos partes mas o menos iguales y eso logra que el árbol quede balanceado lo cual es fundamental para que la búsqueda dentro del mismo sea de orden logarítmico y no lineal.

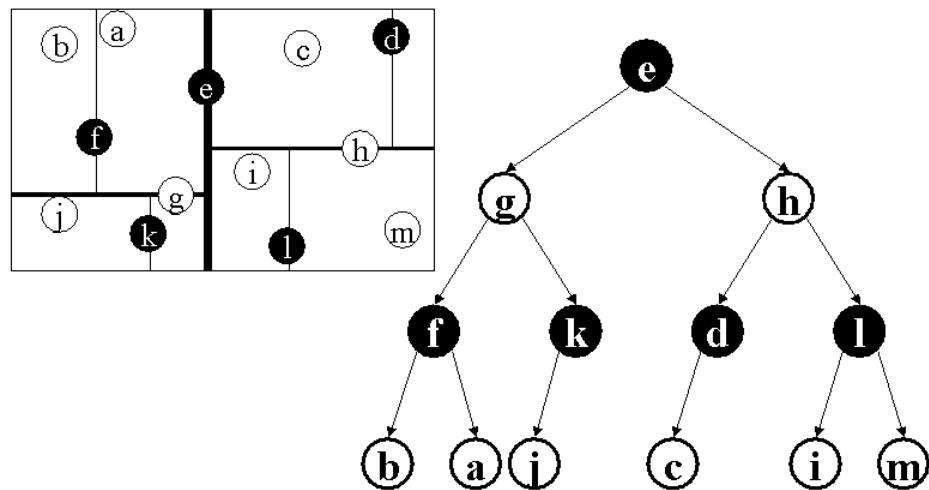


Figure 2.11: KD-Tree en 2D

La figura 2.11 nos muestra un KD-Tree en dos dimensiones y la forma en la cual el árbol partitiona el espacio. La representación que se obtiene se llama "treemap" y es una visualización popular para datos que se presentan de forma jerárquica como los árboles, carpetas en un disco, etc.

Un KD-Tree permite buscar los puntos mas cercanos a un query sin tener que recorrer todos los puntos del set de datos, comenzando por la raíz vamos navegando el árbol de acuerdo a las coordenadas del punto que buscamos, eventualmente hace falta hacer algo de backtracking para obtener los k vecinos mas cercanos al punto pero en general se trata de una operación de orden logarítmico, eso es bueno y en principio podríamos pensar que un KD-Tree o algún otro de los índices espaciales es la solución ideal al problema de la búsqueda de los vecinos mas cercanos para el algoritmo KNN. Sin embargo esto no es así.

El problema de los KD-Trees y de todos los índices espaciales en general es que sólo funcionan bien para muy pocas dimensiones, 2, 3 o 4 dimensiones. A medida que la cantidad de dimensiones aumenta los índices degradan muy rápidamente a punto tal de ser iguales a la simple comparación por fuerza bruta contra todos los puntos. Este problema es una versión del meta-problema conocido como "la maldición de la dimensionalidad" del cual hablaremos mas adelante pero que ya podemos definir de una forma muy simple: "algunos algoritmos no funcionan bien con muchas o muy pocas dimensiones". En este caso los índices espaciales sólo funcionan bien con muy pocas dimensiones. Es por esto

que no les dedicamos mucho espacio ni dedicación ya que es muy raro contar con un problema de Data Science en el cual los datos se presenten en dos o tres dimensiones. Los índices espaciales son muy útiles en aplicaciones gráficas, juegos y todo tipo de programas en los cuales trabajemos en dos o tres dimensiones, lamentablemente no es nuestro caso. Mas adelante al hablar de dimensionalidad vamos explicar mejor el motivo por el cual los KD-Trees degradan a una búsqueda lineal cuando tenemos muchas dimensiones.

2.6.3 Indices Espaciales: VP-Trees

Los VP-Trees son otra forma de índice espacial, también sufren la maldición de la dimensionalidad pero resisten mucho mas a la misma que un KD-tree, es decir que si un índice espacial nos va a ayudar a que KNN funcione mas rápido lo mas probable es que dicho índice sea un VP-Tree.

Un VP-Tree debe su nombre a (Vantage Point Tree) y afortunadamente es una estructura de datos muy sencilla que no necesita de ningún parámetro, el único detalle crucial es que la métrica usada para las distancias en KNN debe cumplir sí o sí con la desigualdad triangular.

Vamos a comentar primero como se construye un VP-Tree y luego como usarlos para resolver el problema de los vecinos mas cercanos a un cierto punto.

Empezamos considerando a todos los puntos en un único nodo. Elegimos al azar un punto que será nuestro "Vantage Point" inicial. Calculamos la distancia desde dicho punto a todos los demás y calculamos la mediana de todas las distancias. Luego dividimos los puntos en dos conjuntos: aquellos cuya distancia es menor o igual a la mediana y aquellos cuya distancia es mayor a la media. La raíz del árbol contiene el Vantage-Point (punto elegido), la mediana de las distancias que llamaremos μ y dos punteros uno al sub-árbol izquierdo con todos los puntos con distancia menor o igual a la mediana y otro al sub-árbol derecho con todos los puntos con distancia mayor a la media. Este proceso se repite recursivamente hasta que queda formado el árbol.

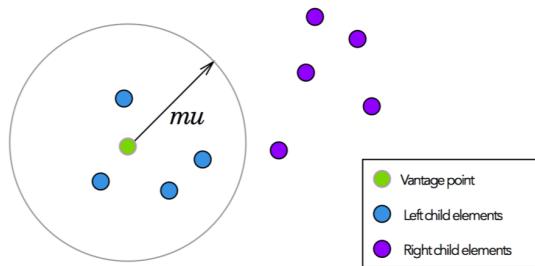


Figure 2.12: VP-Tree

Cuando queremos realizar un query empezamos por la raíz. Si la distancia desde el punto query q a la raíz (τ) es menor a μ entonces calculamos si con

un radio de τ estamos siempre dentro del radio de μ alrededor del punto raíz. Si esto ocurre solo tenemos que explorar la rama izquierda del árbol usando la distancia entre el query y la raíz como valor para τ .

Si con un radio de μ alrededor del punto query quedamos siempre fuera del círculo de radio μ alrededor de la raíz entonces solo tenemos que explorar el sub-árbol derecho usando $\tau = \mu$

En el peor caso estamos en un caso intermedio entre los dos anteriores y tenemos que explorar ambos sub-árboles.

Una vez que accedemos al sub-árbol izquierdo o derecho analizamos la distancia entre el punto query y esta raíz y repetimos el proceso.

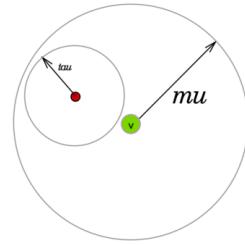


Figure 2.13: VP-Tree caso en el cual solo recorremos el sub-árbol izquierdo

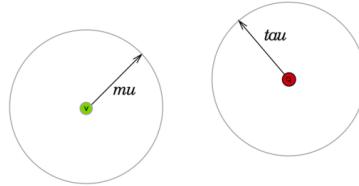


Figure 2.14: VP-Tree caso en el cual visitamos solo el sub-árbol derecho

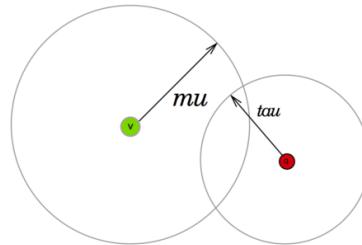


Figure 2.15: VP-Tree caso en el cual visitamos ambos sub-árboles

Los VP-Trees son sencillos de construir y usar y suelen ser los índices espaciales que mejor resisten la maldición de la dimensionalidad. Notemos que el proceso de construcción del árbol es extremadamente simple y que para usar el árbol para resolver queries solo hace falta lógica para determinar en que hijos visitar de cada nodo y cuando podemos detener la búsqueda al ya tener k puntos cercanos al query y estar seguros de que no puede haber mas. El árbol tiene toda la información necesaria para poder hacer esto.

Si bien los VP-Trees son superiores (en general) a los KD-Trees no son perfectos y cuando trabajos con muchas dimensiones puede darse que en todos los casos haya que recorrer los dos hijos de cada nodo lo cual degrada en una búsqueda lineal o peor (por el overhead). Esto ocurre rápidamente en datos sintéticos y aleatorios, incluso con pocas dimensiones pero como los datos reales nunca son random podemos suponer que no se distribuyen de forma uniforme por lo que un VP-Tree puede darnos una ventaja cuando la cantidad de puntos con la que estamos trabajando es realmente muy grande incluso en muchas dimensiones. Esto nos muestra que existe una puja constante entre la maldición de la dimensionalidad y la bendición de la no-uniformidad de los datos y diferentes algoritmos explotan esta tensión de forma diferente.

2.6.4 Líderes y seguidores

El algoritmo de líderes y seguidores es un algoritmo aleatorizado para aproximar el problema de los k vecinos mas cercanos. Este algoritmo necesita de una etapa de pre-procesamiento de los datos que se hace una única vez y luego de esta permite aproximar los vecinos mas cercanos sin necesidad de comparar contra todos los puntos. El algoritmo es bastante sencillo y lo describimos a continuación.

En primer lugar tomamos una cantidad de puntos al azar de nuestro set de datos, en general \sqrt{n} y los denominamos "líderes". Luego procesamos cada uno de los puntos que quedaron del set de datos y comparando contra cada líder lo asignamos al líder mas cercano. De esta forma luego de la etapa de pre-procesamiento cada punto del set de entrenamiento o bien es un líder o bien está asociado (linkeado) a un líder.

Para buscar los k vecinos mas cercanos a un cierto query lo que hacemos es buscar comparar el punto contra cada líder y para el líder mas cercano comparar contra todos sus seguidores. Es decir que hacemos únicamente comparaciones con $2 * \sqrt{n}$ puntos ya que suponemos que cada líder tiene asociados en promedio \sqrt{n} seguidores. Es evidente que este algoritmo es una aproximación ya que puede ocurrir que los k puntos encontrados no sean exactamente los k vecinos mas cercanos, tal vez un punto mas cercano estaba asociado a otro líder y se nos escapa de la búsqueda. Es posible "tunear" la precisión de este algoritmo sacrificando velocidad por mayor precisión, una forma de hacer esto es comparar contra los n líderes mas cercanos en lugar de uno solo.

Algorithm 10: Lideres y seguidores: pre-procesamiento

Data: data:X, labels:Y, number of observations:m
Result: set of leaders

```

1 leaders = randperm(m)[1.. $\sqrt{m}$ ];
2 for point in X do
3   if point not in leaders then
4     compare point against each leader;
5     assign point to closest leader;
```

Algorithm 11: Lideres y seguidores:query

Data: query: Q data:X, leaders:L,k,metric
Result: k closest neighbors

```

1 l = findClosestLeader(Q,L);
2 candidates = findPointsByLeader(X,l);
3 neigbors = KNN(Q, candidates, k, metric);
```

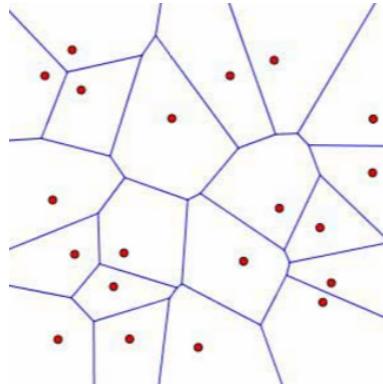


Figure 2.16: Diagrama de Voronoi en 2D representando los centroides de K-Means

2.6.5 Aproximación con K-Means

Una opción muy similar al método de líderes y seguidores es aplicar K-Means a los datos tomando \sqrt{m} centroides (m es la cantidad de puntos). K-Means es un algoritmo de clustering que veremos mas adelante que nos devuelve un conjunto de centroides y a cada punto lo asigna al centroide mas cercano, igual que en líderes y seguidores.

Cuando queremos los k vecinos mas cercanos a un punto comparamos contra los centroides y los puntos que pertenecen a ese cluster. Eventualmente podemos comparar contra los b centroides mas cercanos si queremos mejor precisión con

el costo de algunas comparaciones mas.

Este método difiere de líderes y seguidores en que los líderes no son ahora puntos tomados al azar sino los centroides encontrados por K-Means por lo que es mas probable que se distribuyan mejor en el espacio de nuestros datos. La desventaja es que tenemos que correr K-Means sobre el set de datos lo cual es menos eficiente que simplemente tomar los líderes al azar.

Dependiendo del caso uno u otro método pueden funcionar muy bien para aproximar el problema.

2.6.6 Editing

El proceso conocido como "editing" en KNN consiste en eliminar del set de entrenamiento puntos que no son necesarios para la clasificación de otros puntos. Consideremos que para un cierto valor fijo de k nuestro espacio de puntos definido por el set de entrenamiento queda dividido en áreas que corresponden a cada una de las clases posibles, cualquier punto futuro que caiga dentro de cada área será clasificado con la clase que corresponde al área dentro de la cual ha caído el punto.

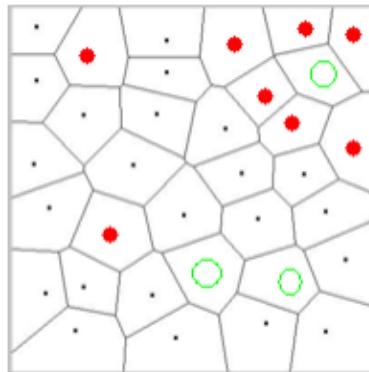


Figure 2.17: KNN fronteras

Notemos que para definir cada área sólo es necesario contar con los puntos que están cerca de las fronteras de las mismas, los puntos interiores no juegan ningún papel en la clasificación de puntos cuya clase desconocemos. El proceso de editing tiene como objetivo eliminar dichos puntos. Los puntos que son realmente relevantes para clasificar los vamos a llamar *prototipos* nuestro objetivo es encontrar los prototipos y descartar el resto de los puntos.

Para encontrar los prototipos hay dos procedimientos, uno parecido a forward selection y el otro a backward selection solo que en lugar de features estaremos seleccionando instancias (puntos).

En forward-edit comenzamos con un conjunto vacío de puntos, luego por cada punto de nuestro set de datos nos fijamos si el punto quedaría correcta-

mente clasificado con los puntos que tenemos hasta el momento, si el punto queda correctamente clasificado entonces lo descartamos mientras que si queda mal clasificado entonces es necesario cambiar la frontera por lo que agregamos el punto al conjunto de prototipos.

Algorithm 12: Forward Editing

Data: data:X,k,metric
Result: T:set of prototypes

```

1 T =
2 for point x in X do
3   if x is NOT correctly classified in T then
4     add point x to T
```

La versión llamada backward-editing comienza con todos los puntos. Luego por cada punto del conjunto verificamos si al removerlo el punto igual quedaría bien clasificado en cuyo caso removemos al punto del conjunto. El resultado final es el set de prototipos.

Algorithm 13: Backward Editing

Data: data:X,k,metric
Result: T:set of prototypes

```

1 T = X
2 for point x in T do
3   if x is correctly classified in T-{x} then
4     remove point x from T
```

El proceso de editing mejora la performance de KNN y no cambia en absoluto la precisión del algoritmo para clasificar ya que solo remueve de nuestro set de entrenamiento aquellos puntos que no son necesarios para clasificar. El costo es la necesidad de pre-procesar todos los puntos para obtener los prototipos tarea que sólo debemos realizar una única vez.

2.6.7 NN vía Grafos

Es posible usar grafos para construir una estructura de datos que nos permita aproximar el problema de los k vecinos mas cercanos de forma eficiente. Para construir el grafo es necesario realizar un pre-procesamiento de los datos. Cada punto será un nodo del grafo y lo vamos a linkear a sus k vecinos mas cercanos. Esto implica hacer una pasada de KNN para todos los puntos del set de entrenamiento, podemos combinar esta pasada con el proceso de editing y eliminar puntos redundantes o ruidosos al mismo tiempo. El resultado del pre-procesamiento es entonces un grafo con tantos nodos como puntos y donde cada nodo tiene k aristas.

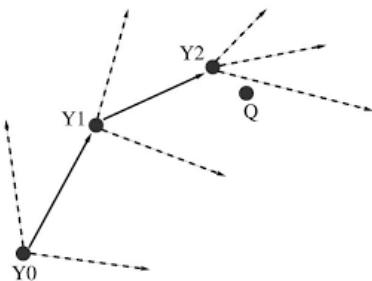


Figure 2.18: KNN aproximado por grafos

Para buscar los k puntos mas cercanos a un cierto query lo que hacemos es comenzar desde un punto al azar del grafo y calcular la distancia entre el query y los vecinos de dicho punto. Una vez que hicimos esto nos movemos al nodo que mas nos acerca al query. Este proceso lo repetimos hasta que no hay posibilidad de acercarnos mas al query. En el camino nos quedamos con los k puntos que hayan estado mas cerca del query que estamos procesando. En la figura 2.18 si empezamos en Y_0 y nuestro Query es Q nos movemos primero a Y_1 y luego a Y_2 guardando siempre en el camino cuáles son los k puntos mas cercanos a Q de los que hemos analizado. Por cada nodo visitado siempre analizamos sus k vecinos. Este algoritmo es bastante rápido y eficiente siempre y cuando sea factible realizar el pre-procesamiento necesario.[Dong][Hajebi]

2.6.8 LSH

LSH (Locality Sensitive Hashing) es un conjunto de algoritmos para determinar rápidamente datos que son similares entre sí, la idea es usar una función de hashing especial de forma tal que si dos datos son similares la función de hashing los asigne a una misma posición o bucket. Es fácil entender cómo estas funciones pueden usarse para aproximar el algoritmo KNN en lo que denominaremos ANN (Approximate Nearest Neighbors), por cada punto simplemente aplicamos la función de hashing y comparamos contra todos los puntos que estén dentro del bucket indicado. Esto permite aproximar los k vecinos mas cercanos a un cierto punto en $O(1)$ lo cual es extremadamente eficiente. LSH puede usarse con diferentes métricas y es un tema tan importante que mas adelante le dedicaremos un capítulo completo. Por el momento sepamos que LSH es una de las formas mas eficientes para aproximar el problema de los vecinos mas cercanos a un punto.

2.7 Teoría de KNN

”Here be dragons”

Esta es la sección mas teórica del capítulo en donde vamos a desarrollar el error que comete KNN al clasificar. Para empezar debemos plantear cuál es el error mínimo que puede tener un clasificador que llamaremos e^* , para entender este error supongamos que tenemos un set de datos en el cual todos los puntos tienen exactamente los mismos atributos pero diferente clase. Esta claro que lo mejor que podemos hacer es $1 - P(C)$ siendo "C" la clase mayoritaria. Si tenemos dos clases, una con el 70% de los puntos y otra con el 30% entonces nuestro error será de 0.3. El error siempre lo vamos a calcular como un número entre 0 y 1 que es la probabilidad de que clasifiquemos mal un punto. Si todos los puntos son iguales lo mejor que podemos hacer es predecir la clase mayoritaria para todos los puntos y nuestro error será igual a la proporción de puntos que no están en esta clase. Vamos a analizar ahora cuál es el error cometido por KNN cuando $k = 1$ es decir que clasificamos únicamente en base al vecino mas cercano, llamemos a este algoritmo NN. (por nearest neighbor).

2.7.1 Teorema de Cover-Hart

Theorem 2 (Cover-Hart) *Si e^* es el error óptimo de un clasificador entonces si tenemos infinitos datos el error de NN es $e_{NN} \leq 2e^*$*

Supongamos únicamente dos clases (+) y (-).

$$e_{NN} = P^+ * P_{NN}^- + P^- * P_{NN}^+$$

Dejando todo en función de los positivos:

$$e_{NN} = P^+ * (1 - P_{NN}^+) + (1 - P^+) * P_{NN}^+$$

Es decir que el error cometido por NN es igual a la probabilidad de que el punto sea positivo por la probabilidad de que NN lo clasifique como negativo mas la probabilidad de que el punto sea negativo por la probabilidad de que NN lo clasifique como positivo.

Veamos qué pasa cuando tenemos infinitos puntos es decir $n \rightarrow \infty$

$$\begin{aligned}\lim_{n \rightarrow \infty} P_{NN}^+ &= P^+ \\ \lim_{n \rightarrow \infty} P_{NN}^- &= P^-\end{aligned}$$

Esto vale ya que con infinitos puntos la probabilidad que NN califique a un punto como positivo es exactamente igual a la proporción de puntos positivos en el set de datos que es P^+

Por lo tanto con infinitos puntos podemos reescribir el error de NN como:

$$e_{NN} = P^+ * (1 - P^+) + (1 - P^+) * P^+$$

Supongamos ahora que P^+ es la clase minoritaria, ya que es lo mismo suponer que es P^+ o P^- . Si P^+ es la clase minoritaria entonces $e^* = P^+$ y podemos entonces escribir el error de NN en función de e^*

$$e_{NN} = 2e^*(1 - e^*) \leq 2e^*$$

Esta última fórmula es la desigualdad de Cover-Hart y es muy importante ya que nos asegura que NN tiene en el peor de los casos el doble de error que un clasificador ideal, esto puede parecer mucho si el error ideal es 0.3 porque NN podría llegar a errar en el 60% de los casos pero cuando el error ideal es bajo como por ejemplo 0.01 NN no puede ser peor que 0.02.

Una conclusión interesante de la desigualdad es que si el error óptimo es e^* y el error de KNN con $k = 1$ es a lo sumo $2e^*$ entonces cuando tomamos $k > 1$ nuestro mejor resultado es a lo sumo el doble de tomar $k = 1$. En otras palabras si con $k = 1$ tenemos un error de 0.14 entonces lo mejor que podemos obtener con cualquier otro valor de k es un error de 0.07. Esta es una cota extremadamente importante ya que permite probar métricas con $k = 1$ y luego analizar cuánto pueden mejorar tomando mayor cantidad de vecinos.

La desigualdad de Cover-Hart puede extenderse a KNN y lo que obtenemos es que para KNN el error es a lo sumo el error ideal multiplicado por una constante "c" muy chica, esto quiere decir que si los datos fueran infinitos KNN sería óptimo. Hay que tener cuidado al interpretar esto ya que los datos **nunca** son infinitos y por más grande que sea el conjunto de datos la diferencia entre infinito y un número muy grande es infinito. Por lo tanto no podemos decir que con muchos datos KNN sea óptimo.

La conclusión que sí podemos obtener es que KNN mejora a medida que tenemos mayor cantidad de puntos, cuanto mejor definidas queden las fronteras mejor será la performance del algoritmo en clasificar puntos nuevos.

2.8 Parzen Windows

El algoritmo Parzen Windows es un familiar muy cercano de KNN, en lugar de seleccionar siempre los k vecinos más cercanos Parzen Windows selecciona los vecinos que están dentro de una cierta distancia σ prefijada. Esto lleva a que la cantidad de vecinos a considerar sea variable.

La figura 2.19 muestra la diferencia existente entre Parzen Windows y KNN, en uno es variable la cantidad de vecinos y fija la distancia y en el otro se fija la cantidad de vecinos y las distancias son variables. En cuanto a los resultados obtenidos en general ambos algoritmos dan resultados bastante similares, Parzen Windows suele usarse más para problemas de regresión que para clasificación aunque los motivos teóricos para que esto ocurra nunca han sido expuestos.

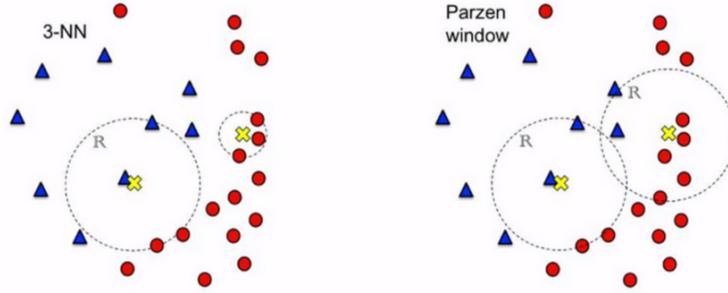


Figure 2.19: KNN vs Parzen Windows

2.9 KNN con pesos

Una extensión lógica al algoritmo KNN es darle pesos a los vecinos de acuerdo a su proximidad con respecto al punto que queremos estimar. Esto es especialmente lógico en problemas de regresión en donde podemos razonar que si nuestro punto está realmente muy cerca de otro entonces el valor a estimar debería ser muy parecido al de dicho punto sin que vecinos mas alejados influyan mucho. Notemos que el uso de pesos invalida el proceso de editing ya que ahora todos los puntos del set de entrenamiento pueden ser útiles.

La idea es entonces asignar a cada uno de los k vecinos un peso W_i que vamos a calcular de la siguiente forma:

$$W_{i \neq 1} = \frac{d(x, x_k) - d(x, x_i)}{d(x, x_k) - d(x, x_1)}$$

Con la salvedad de que el punto mas cercano tiene siempre peso igual a 1 y el mas lejano tiene peso 0. x_k es el vecino mas lejano, x_1 es el vecino mas cercano.

Por ejemplo supongamos que $k = 5$ y las distancias que tenemos son (ordenadas de menor a mayor): (3,3.25,6,7,13).

Para el primer punto el peso es igual a 1 por definición. Para el segundo punto el peso lo calculamos de la forma:

$$W_2 = \frac{13 - 3.25}{13 - 3} = 0.975$$

Mientras que para el cuarto punto el peso es:

$$W_4 = \frac{13 - 7}{13 - 3} = 0.6$$

Una vez que calculamos los pesos es sencillo estimar el valor de regresión para el punto en cuestión como:

$$\bar{Y} = \frac{\sum_i W_i * X_i}{\sum_i W_i}$$

Notemos que también podemos usar este esquema para problemas de clasificación. En KNN antes cada vecino tenía un voto para determinar a que clase pertenece el punto, ahora tendrá el valor correspondiente a W_i y la clase que sume mas es la que determina la clase del nuevo punto.

Otra forma popular de "pesar" los vecinos en KNN es mediante la fórmula:

$$W_i = \frac{1}{d(x, x_i)^\beta}$$

En donde β es un parámetro. Cuando $\beta = 0$ tenemos el algoritmo tradicional, todos los puntos tienen peso = 1. Cuando $\beta = 1$ tenemos una interpolación lineal, cuando $\beta = 2$ le damos un peso cada vez menor a los puntos que se alejan del punto original.

Finalmente una tercera opción es pesar los puntos mediante un gaussiano alrededor de cada punto:

$$W_i = \exp\left(-\frac{|x_i - x|^2}{2\sigma^2}\right)$$

σ es el radio alrededor de cada punto para considerar que los puntos dentro del mismo son vecinos. Podemos determinar σ calculando el promedio de las distancias entre cada punto y su k-ésimo vecino mas cercano.

Un detalle muy interesante es que al usar pesos podemos usar un valor de k tan grande como deseemos, los puntos mas lejanos simplemente van a tener un peso cada vez menor y en concreto podríamos usar $k = n$ en cuyo caso para clasificar un punto tenemos en cuenta todos los puntos del set de datos como vecinos. Cuando esto pasa lo que tenemos es efectivamente un *kernel* de distancias entre puntos. Mas adelante al ver otros algoritmos de clasificación dedicaremos un buen tiempo al estudio de métodos basados en kernels.

Existen otros esquemas interesantes para usar pesos en KNN descritos en [Gou]

2.10 Evitando Overfitting en KNN

En KNN, y especialmente cuando k es un número chico, existe el riesgo de overfitting, es decir que el algoritmo funcione muy bien para el set de entrenamiento pero no tan bien para puntos nuevos, esto es un error de generalización.

Una forma de evitar el error de generalización consiste en eliminar del set de entrenamiento puntos anómalos definiendo como punto anómalo aquel para el cual todos sus vecinos pertenecen a una clase diferente a la del punto. Luego de eliminar los puntos anómalos la capacidad de generalizar del algoritmo debería ser superior ya que se elimina el efecto de estos puntos en la frontera de algoritmo. Este proceso es compatible con el proceso de edición, es posible al mismo tiempo que buscamos los prototipos eliminar los puntos anómalos.

2.11 RKNN: Ensambls basados en KNN

Es posible combinar el resultado de varios algoritmos KNN en un ensamble. Hemos mencionado que KNN es muy sensible a cuáles son los atributos (features) que usamos para calcular las distancias, podemos entonces crear un ensamble en donde cada KNN usará un conjunto de m atributos al azar de nuestro set de datos.

Por ejemplo si tenemos 5 atributos: (A,B,C,D,E) si usamos $m = 3$ podemos tener un KNN que use (A,C,D) otro con (B,C,E) otro con (A,B,D), etc. Cada KNN va a realizar una predicción para el set de test o bien para cada punto de nuestro set de datos (leave one out cross validation), la predicción del ensamble surge de cuál es la opinión mayoritaria de todos los KNN. En un problema de clasificación binaria si tenemos un total de 100 KNNs y 82 de ellos dicen que un punto es de clase "1" y los otros 18 dicen que es de clase "0" entonces vamos a clasificar al punto como de clase "1" con probabilidad 0.82.

La cantidad de atributos m es un hiper-parámetro al igual que k y la métrica a usar como distancia, los tres hiper-parámetros tienen que buscarse por grid-search usando cross-validation.

El uso de un ensamble permite minimizar el impacto de atributos que no son buenos predictores o que afectan el resultado de KNN, un detalle muy importante es que el ensamble nos permite calcular la importancia predictora de cada atributo. Cada KNN tiene un cierto promedio de precisión, que podemos extenderle a cada uno de los atributos que participan del mismo, por lo tanto podemos promediar el promedio de precisión de cada atributo simplemente calculando la precisión de los KNN en los cuáles el atributo participa. Por ejemplo:

Atributos	Precisión
A,C,D	0.78
A,B,E	0.84
B,C,D	0.81
C,D,E	0.75

Table 2.4: Ejemplo de un ensamble de 4 KNNs con 3 atributos sobre un total de 5

En este caso el atributo "A" tiene una precisión promedio de 0.81, "B" tiene 0.825, "C" tiene 0.78 y "D" tiene 0.78. Esto lo podemos usar para detectar cuáles son los atributos que ayudan mas a KNN y cuáles son los menos significativos y eliminarlos, realizando un nuevo ensamble con menor cantidad de atributos posibles. Este es un caso de feature-selection basado en un ensamble de KNNs.

2.12 Algunas conclusiones y comentarios

KNN es un algoritmo extremadamente importante. Es capaz de funcionar tanto para problemas de clasificación como para problemas de regresión y se ajusta perfectamente a fronteras no-lineales. Requiere el ajuste de únicamente dos

hiper-parámetros que son el valor de k (si no usamos pesos) y la métrica a usar para determinar las distancias. A mayor cantidad de puntos en el set de entrenamiento mejor es el resultado de KNN y en el límite si los puntos fueran infinitos KNN es óptimo en el sentido de minimizar el error de clasificación. Irónicamente KNN, en su versión mas *naive* escala muy mal al tener que calcular las distancias contra todos los puntos del set de datos por cada punto a clasificar. Vimos varios algoritmos y métodos para hacer que KNN sea mas eficiente ya sea reduciendo la cantidad de puntos contra los cuales tenemos que calcular las distancias o usando algoritmos de aproximación que sólo usan un subconjunto del set de entrenamiento para clasificar. Los índices espaciales que funcionan muy bien en pocas dimensiones degradan en la búsqueda por fuerza bruta en espacios de dimensiones medianas a grandes por lo que en general no los usamos para KNN, nos referimos a esto como un caso de la maldición de la dimensionalidad y este es el tema de nuestra próxima sección.

2.13 Dimensionalidad

Estudiar la dimensionalidad de los datos es un paso fundamental, imprescindible en la mayoría de los procesos de Data Science. Este estudio implica entender en que espacio se presentan los datos, en que espacio residen realmente, cómo se comportan los algoritmos que pretendemos usar en estos espacios y si resulta conveniente realizar alguna transformación del espacio original a otro espacio a efectos de lograr algún beneficio.

2.13.1 Manifolds

Un Manifold es un espacio que localmente se comporta como un espacio euclídeo aunque globalmente no lo sea. El mejor ejemplo que podemos dar para ilustrar esto es el de una esfera como nuestro planeta mismo. En una esfera la distancia euclídea no funciona, sin embargo una pequeña porción de la esfera puede tratarse como un espacio euclídeo y es así como medimos las distancias con las cuales nos manejamos diariamente. Para ir desde un punto de una ciudad a otro nadie tiene en cuenta que se está moviendo sobre la superficie de una esfera.

Un ejemplo de algo que no es un Manifold es la figura de ocho que mostramos en el centro de la figura 2.20. En este espacio los puntos en el "cruce" nunca se comportan como un espacio euclídeo sin importar que tan chica sea la escala en la cual se trabaje.

En muchos casos un conjunto de datos se presenta en un espacio que no corresponde a la dimensionalidad real de los datos, cuando tenemos un espacio representado en otro hablaremos de un "embedding", este puede ser de mayor o menor cantidad de dimensiones que el espacio original. La espiral a la derecha de la figura 2.20 es un buen ejemplo de esto, lo que tenemos es un Manifold unidimensional en un espacio de dos dimensiones. Si tomamos una pequeña porción de la espiral los puntos se comportan localmente bajo las reglas de un

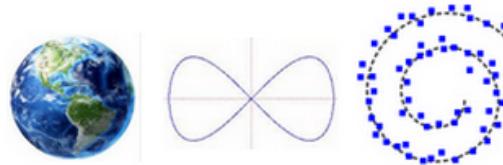


Figure 2.20: Manifolds

espacio euclídeano y podríamos estirar la espiral hasta quedarnos con una recta que representaría a nuestros datos perfectamente. A los algoritmos que intentan descubrir la verdadera dimensionalidad de los datos los llamaremos algoritmos de "Manifold Learning", una categoría muy importante dentro de Data Science.

2.13.2 Los Datos No son Random y Siempre Tienen Pocas Dimensiones

Este concepto es fundamental y muchas veces se lo ignora lo cual lleva a realizar todo tipo de afirmaciones incorrectas e incluso a predecir de forma totalmente infundada el comportamiento de un algoritmo. La premisa que intentaremos explicar es que independientemente de la cantidad de dimensiones en las cuales se presenten los datos estos casi siempre tienen pocas dimensiones. Es decir que frecuentemente nuestros datos se van a presentar como un manifold de pocas dimensiones inmerso en un espacio dimensional mucho mayor. Algo semejante a lo que sería una superficie de dos dimensiones en un espacio tridimensional.

Para explicar por qué decimos que los datos siempre tienen pocas dimensiones la clave es entender que los datos no son aleatorios, si fueran aleatorios no serían datos sino ruido. Que los datos tengan un significado, que sean entendibles por un ser humano implica que existe una estructura en los mismos y esto implica que no sean aleatorios. Como los datos no son aleatorios es imposible que "cubran" todo el espacio dimensional en el cual se presentan. Empecemos con un ejemplo muy simple: Datos sobre personas para las cuales registramos la altura y peso. Estos datos se presentan en 2d pero no ocupan el espacio total en 2d, por ejemplo una persona que mide 1 metro 50 no puede pesar 1345kg y una persona que pesa 55kg no puede medir 4 centímetros.

Otro ejemplo muy claro son las imágenes, supongamos que tenemos un set de datos con imágenes de 32x32 pixeles en blanco y negro. Está claro que nuestros datos se presentan en un espacio de 1024 dimensiones (32x32) pero también debe quedar claro que nuestras imágenes de ninguna forma cubren la totalidad de dicho espacio, esto es porque no cualquier vector de 1024 pixeles es una imagen válida.

Las imágenes al azar que mostramos no tienen ningún sentido y nunca for-

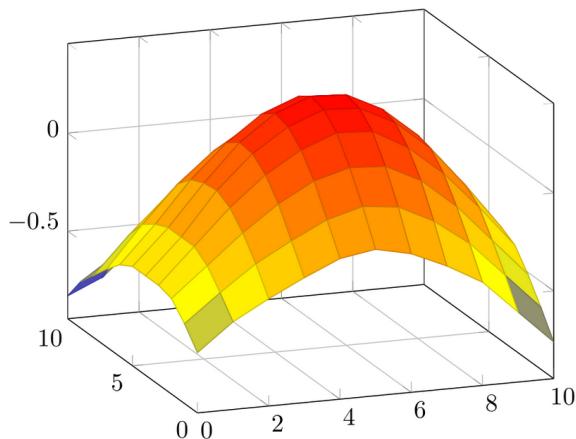


Figure 2.21: Manifold de 2d en 3d

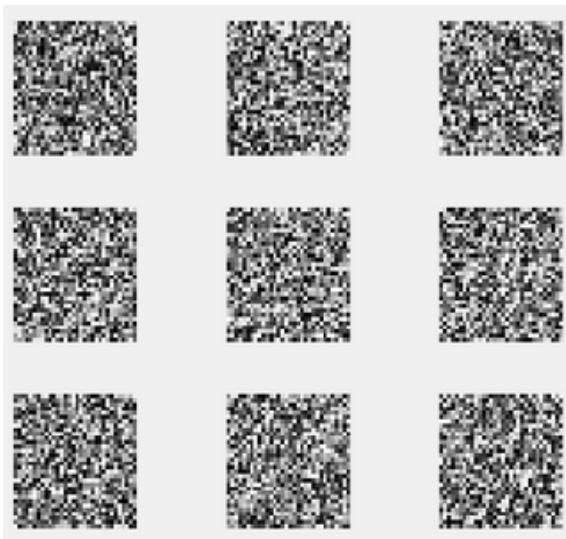


Figure 2.22: Imágenes aleatorias

marián parte de nuestro set de datos, por ejemplo si las imágenes fueran fotos podríamos argumentar que no son fotos posibles. Esto quiere decir que sólo un subespacio de nuestro espacio de 1024 dimensiones contiene las imágenes que serían válidas, aquellas que tienen sentido. En otras palabras las imágenes se presentan en un manifold de una cierta cantidad de dimensiones inmerso en un espacio de 1024 dimensiones.

2.13.3 Manifold Learning y Cambios de Dimensiones

Los algoritmos de manifold learning y cambio de dimensiones permiten transformar los datos de un espacio dimensional a otro, para hacer esto tiene que existir algún motivo válido, es decir una razón que explique por qué nos conviene trabajar en un espacio de dimensiones que no es aquel en el cual se presentaron los datos originalmente. Algunos motivos muy comunes son:

1. Para poder visualizar los datos en dos o tres dimensiones. (este es un motivo poderoso!)
2. Porque el algoritmo que queremos usar funciona mejor en otro espacio dimensional. (combatir la maldición de la dimensionalidad)
3. Por razones de eficiencia de tiempo o espacio
4. Para eliminar el ruido de nuestro set de datos

En mayor o menor medida es probable que uno o varios de estos motivos se presenten en nuestras aplicaciones. Es por esto que casi siempre vamos a tener que usar de una forma u otra algún algoritmo de cambio de dimensiones. Existen muchos algoritmos para cambiar las dimensiones de los datos y vamos a dedicarles un capítulo entero mas adelante.

2.13.4 La Maldición de la dimensionalidad

Este tema es muy popular y lamentablemente en una enorme cantidad de sitios, blogs, apuntes e incluso libros está incorrectamente definido y explicado, vamos a intentar explicar de qué se trata realmente la "maldición" y por qué se presta a confusión.

La definición que vamos a usar para la maldición de la dimensionalidad es muy simple: "no todos los algoritmos se comportan bien en cualquier espacio de dimensiones". Algunos algoritmos se comportan bien en pocas dimensiones y otros se comportan mejor en muchas dimensiones. Hemos estudiado KNN y hemos visto que si trabajamos en pocas dimensiones podemos usar un índice espacial, podríamos decir entonces que la maldición de la dimensionalidad para KNN pasa por no poder usar índices espaciales y tener que caer en una búsqueda por fuerza bruta u otros recursos cuando las dimensiones son muchas. Es importante entender que para cada algoritmo hay razones completamente diferentes por las cuales podemos preferir muchas o pocas dimensiones.

Habiendo hecho esta definición tan genérica y mostrado un ejemplo conocido que ya hemos estudiado podríamos dar por cerrado el tema, sin embargo, es conveniente mostrar algunos efectos de tener muchas o pocas dimensiones y cómo a veces estos efectos pueden llevar a alguna confusión acerca de su influencia en el funcionamiento de los algoritmos.

El problema del muestreo

En todo problema de Data Science es correcto decir que cuantos mas datos podamos recolectar mejor serán nuestros resultados. Esto es algo lógico ya que los datos ayudan a que nuestros algoritmos puedan entenderlos y generar mejores resultados. En 2001 Banko y Brill [Banko,Brill] publicaron un trabajo que se hizo muy famoso en el cual sostenían que en la mayoría de los casos recolectar mas datos era mas importante que usar un algoritmo mejor. Dicho de otra forma con cantidad de datos suficientes incluso algoritmos muy simples convergen al mismo resultado que los algoritmos mas avanzados.

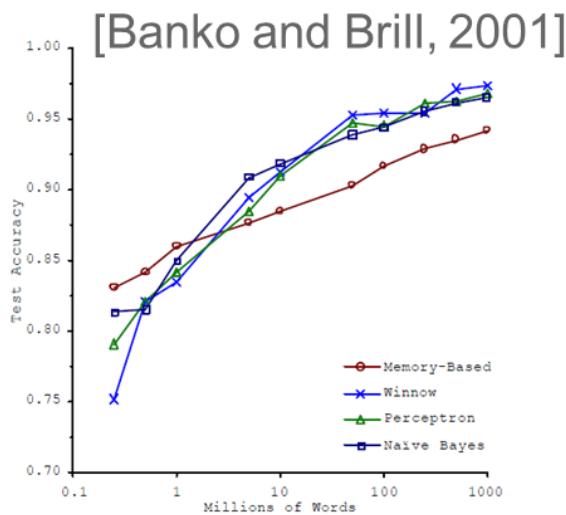


Figure 1. Learning Curves for Confusion Set Disambiguation

Figure 2.23: Banko and Brill 2001

Una vez que hemos establecido que a mayor cantidad de datos mejores resultados tenemos que analizar el efecto de la dimensionalidad en la cantidad de datos necesarios. Es fácil darse cuenta que a medida que aumenta la cantidad de dimensiones del set de datos aumenta exponencialmente la cantidad de datos "posibles" por lo tanto a mayor cantidad de dimensiones necesitamos exponencialmente mas datos para mantener el mismo nivel de muestreo (sampling).

Podemos decir entonces que este es uno de los efectos de la maldición de la dimensionalidad sin embargo tenemos que ser muy cuidadosos, es cierto que a mayor cantidad de dimensiones la cantidad de datos "posibles" aumenta exponencialmente si y sólo si cualquier dato es "posible" pero esto implicaría datos aleatorios y hemos establecido que los datos nunca son aleatorios. Por lo tanto

hay que tener mucho cuidado al decir que a mayor cantidad de dimensiones necesitamos muchos mas datos para tener un buen muestreo, en la mayoría de los casos con datos reales esta afirmación no es válida.

2.13.5 El Efecto de la Dimensionalidad sobre las Distancias

Es necesario estudiar el efecto que la dimensionalidad tiene sobre las distancias, en concreto sobre la distancia general de Minkowsky que definimos como:

$$d(x, y) = (\sum |x - y|^p)^{1/p}$$

Como vivimos en un mundo tridimensional algunos efectos que ocurren al tomar distancias en espacios de muchas dimensiones pueden parecer completamente contra-intuitivos, algunos ejemplos de estos curiosos efectos son:

- En un espacio de muchas dimensiones si tenemos una distribución gaussiana multinomial la mayoría de la masa no está en la mediana de la distribución sino en una cáscara alrededor de la misma.
- En muchas dimensiones la mayor parte del volumen de una hiper-esfera está en su superficie y no dentro de la misma (!)
- En muchas dimensiones los puntos dentro de un hipercubo están mas cerca de la frontera del hipercubo que de su vecino mas cercano
- En muchas dimensiones la distancia euclídea entre puntos tiende a converger

Estos curiosos efectos nos llevan a la conclusión de que debemos tener mucho cuidado al trabajar en espacios multidimensionales y a pensar que la selección de una métrica adecuada puede ser fundamental, por ejemplo para el funcionamiento de un algoritmo como KNN que está basado en distancias.

El efecto de la dimensionalidad en las distancias queda evidenciado por el siguiente teorema:

Theorem 3 (Beyer)

$$\text{if } \lim_{d \rightarrow \infty} \text{var}\left(\frac{\|x\|_d}{E[\|x\|_d]}\right) = 0, \text{ then } \frac{D_{\max} - D_{\min}}{D_{\min}} \rightarrow 0$$

La demostración de este teorema puede encontrarse en [Beyer] y lo que nos dice es que a medida que la cantidad de dimensiones tiende a infinito la diferencia entre la distancia máxima entre dos puntos del espacio y la distancia mínima converge es decir que todas las distancias son aproximadamente iguales. Este es un resultado muy perturbador. Afortunadamente, como hemos visto, los datos reales no suelen ocupar completamente el espacio en el cual se presentan pero de todas formas es importante considerar que en muchas dimensiones el concepto de distancia entre puntos empieza a peligrar.

Consideremos el caso de la norma l_∞ es decir la diferencia máxima que existe entre x e y en alguna dimensión. Esta norma es útil en dos o tres dimensiones pero es claro que a medida que aumentamos la cantidad de dimensiones su utilidad va decreciendo, en un espacio de 100 dimensiones por ejemplo la diferencia máxima en solo una de estas dimensiones ciertamente no debe ser la mejor forma de medir la distancia entre dos puntos. La intuición es que a medida que aumentamos la cantidad de dimensiones exponentes cada vez mas bajos nos dan una mejor medida de distancia entre los puntos, este razonamiento es cierto y fue demostrado tanto en forma teórica como empírica en [Aggarwal,Hinnenburg,Keim]. Se demuestra que para espacios de muchas dimensiones la distancia Manhattan ($p = 1$) funciona mejor que la distancia euclídea e incluso distancias con $p < 1$ funcionan aun mejor.

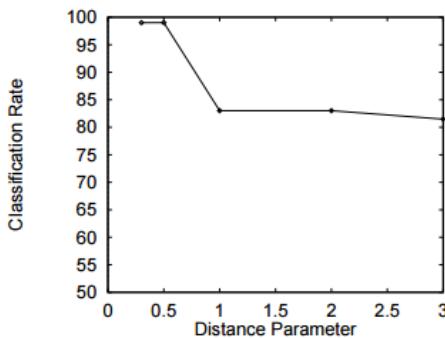


Figure 2.24: Comportamiento de distancias en un algoritmo de clasificación para 20 dimensiones

La conclusión es que en espacios multidimensionales debemos considerar distancias con exponentes bajos, incluso menores a 1 y darles la oportunidad de competir contra la distancia euclídea comparando los resultados porque en muchos casos nos vamos a encontrar con que las distancias con exponentes fraccionarios y menores a uno tienen un rendimiento superior en algoritmos de clasificación y clustering.

El problema de los KD-Trees

Retomamos ahora el problema de los KD-Trees en muchas dimensiones, hemos dicho que en muchas dimensiones un KD-Tree degrada en una búsqueda por fuerza bruta, vamos a intentar una explicación intuitiva de por qué sucede esto.

Nuestro espacio de d dimensiones se puede ver como un hipercubo, consideremos una hiper-esfera inscripta dentro del hipercubo. A medida que la cantidad de dimensiones aumenta el volumen de la esfera se vuelve insignificante con respecto al volumen del hipercubo. Esto es muy fácil de ver pensando en que

porcentaje de superficie ocupa un círculo inscripto dentro de un cuadrado y que porcentaje de volumen ocupa una esfera inscripta dentro de un cubo.

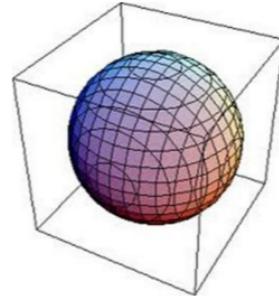


Figure 2.25: Distancias y dimensionalidad

Esta observación sirve para sacar algunas conclusiones interesantes: cuando trabajamos en muchas dimensiones la mayoría de los puntos se concentra en las esquinas del hipercubo y por lo tanto la distancia entre los puntos se vuelve muy parecida. Este es el motivo por el cual un KD-tree se degrada, como las distancias son todas muy parecidas no es posible elegir un split en cada dimensión de forma tal que estemos seguros que el punto mas cercano a nuestro query está de un lado del split y no del otro. En forma intuitiva es como si en cada split del KD-Tree no pudiéramos tomar una decisión ya que para un punto dado el punto mas cercano podría estar en cualquier dirección, por lo tanto no queda otra que considerar prácticamente todos los puntos y por eso los KD-Trees y cualquier índice espacial degradan cuando tenemos muchas dimensiones.

Shared Neighbors

Además del uso de exponentes fraccionarios hay otros trucos que podemos probar cuando trabajamos en muchas dimensiones, uno de estos trucos es el concepto de "vecinos mas cercanos compartidos" o (Shared nearest neighbors). Esta métrica define la distancia entre dos puntos x e y como la intersección entre los vecinos mas cercanos de ambos puntos.

$$SNN(x, y) = |KNN(x, k) \cap KNN(y, k)|$$

Esta métrica ha demostrado funcionar mejor que otras en espacios de muchas dimensiones. [Houle]

Chapter 3

Almacenamiento Distribuido

The world is one Big Data
problem

Andrew McCafee

3.1 Almacenamiento Distribuido

El almacenamiento distribuido consiste, como su nombre lo indica, en almacenar información en varios equipos. En general se utilizan equipos económicos muy similares a un equipo de escritorio con discos comunes. Como este tipo de hardware tiene una cierta tasa de fallas la información se almacena replicada de forma tal que la caída de un equipo no implique la perdida de información. En cada equipo del cluster existe un file-system local que sólo es accesible desde el equipo y un file-system distribuido que es global a todo el cluster. Es posible mover archivos desde un file-system al otro.

En un file-system distribuido un archivo se representa como una sucesión de registros que pueden tener tamaño y estructura arbitrarios, el file-system se encarga de dividir cada archivo en "bloques" y almacenar estos bloques en diferentes equipos del cluster de forma replicada.

Por ejemplo, si tenemos un archivo de 10tb, el file-system puede decidir particionarlo en bloques de 100Mb y almacenar cada bloque en tres equipos diferentes del cluster. Es necesario que las réplicas no estén dentro del mismo rack o switch para que la caída de un switch no implique perder el acceso a esta información.

Todos estos datos son almacenados y manejados por un "name server" o "master node" que es un nodo especial del cluster que "sabe" de que forma se han repartido los archivos en el cluster. El master node se replica en un backup

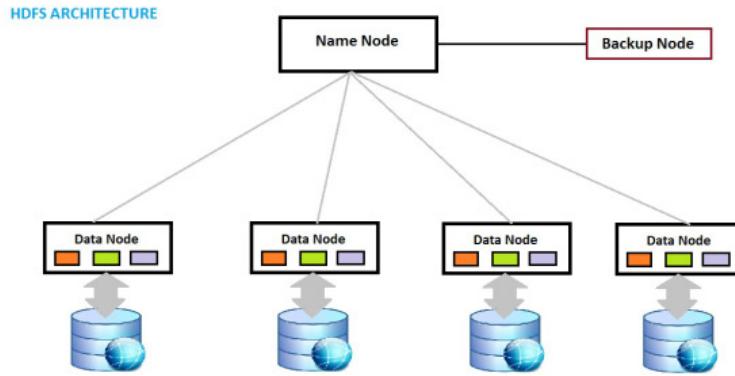


Figure 3.1: HDFS.

de forma tal que la caída de este equipo no implique una perdida catastrófica de todo el file-system distribuido.

Cuando se quiere acceder a un determinado registro de un archivo el "master" se encarga de determinar en que equipos está dicha información y acceder a la misma, si el equipo en cuestión no responde el master se encarga de acceder a una replica y disparar una nueva replica en otro nodo del cluster. Todo esto por supuesto es configurable y dependiente de las diferentes implementaciones y versiones que existen de un file-system distribuido.

Algo importante a destacar es que en general los archivos distribuidos son inmutables, puede leerse y puede agregarse información al final del archivo pero la información existente no puede modificarse. Para modificar un archivo distribuido hay que crear un nuevo archivo distribuido con los cambios necesarios y destruir el anterior.

3.2 Map Reduce

El paradigma de programación Map-Reduce es una abstracción inspirada en el paradigma de programación funcional para procesar información que se encuentra almacenada de forma distribuida en un cluster. La principal característica de este paradigma es que podemos dividir el procesamiento de la información distribuida en tres fases:

1. Fase de Map
2. Fase de Shuffle & Sort
3. Fase Reduce

Vamos a presentar entonces cada una de estas fases de forma abstracta para luego poder usar lo aprendido en ejemplos mas concretos.

3.2.1 Fase de Map

En la fase de Map se realizan todas las operaciones que puedan hacerse de forma atómica sobre cada registro. En otras palabras cualquier cosa que no necesite mas de un registro del archivo. Por ejemplo podemos realizar transformaciones de datos, convirtiendo cada registro del archivo en el formato que necesitemos y creando un nuevo archivo distribuido. Podemos también filtrar datos eliminando los registros que no nos sirven o lo que es igual seleccionando los registros que nos interesan.

Para la fase de Map lo que el "sistema" hace es asignar a cada bloque (conjunto de registros) del archivo una tarea que ejecute el código necesario para realizar la tarea encomendada, si algunos de estos workers falla o no responde el sistema puede asignar otro a la réplica del mismo bloque en otro equipo. En general el usuario no controla la cantidad de procesos "Map" que el sistema utiliza. La salida de un Map puede convertirse en un nuevo archivo distribuido o bien usarse como input para otro proceso Map o un proceso reduce.

La definición "formal" de Map que hace Hadoop es la siguiente:

$$Map(k, v) \rightarrow [k2, v2]$$

Map recibe un registro en formato clave (k) y valor (v) y devuelve una lista de registros en formato (clave,valor). Es de destacar que las claves que Map devuelve no tienen por qué ser las mismas que recibe, ni siquiera tienen que ser parecidas. Por cada registro procesado Map puede devolver uno, muchos o ningún registro.

Desde el punto de vista del usuario/programador Map procesa un registro por vez, desde el punto de vista de Map Reduce el sistema asigna a un proceso Mapper a un cierto bloque del archivo y este proceso se encarga de procesar todos los registros dentro del bloque llamando a la función o método Map definido por el usuario. En definitiva Map tiene que ser una función definida por el usuario que reciba como parámetro un registro y devuelva 0 a N registros en cualquier formato que sea conveniente.

Por ejemplo podemos tener un archivo con números y mediante Map convertirlo en un archivo que contenga únicamente números primos. En este caso la "clave" de cada registro es simplemente un número de registro u algo que podemos ignorar. Por lo que podemos definir Map de la siguiente forma:

$$Map(r) \rightarrow [r']$$

Esta definición es mas genérica que la anterior. Map recibe un registro (r) y devuelve 0 a N registros r'. El registro "r" puede tener cualquier formato, por ejemplo puede ser una tupla de tipo (clave,valor) o puede ser simplemente un valor atómico o cualquier tipo de estructura, por ejemplo un diccionario. Esta

es la definición que vamos a usar para Map ya que es mas genérica que la que vimos basada en Hadoop.

3.2.2 Map Partitions

Map se aplica a cada registro de un archivo distribuido. Un archivo distribuido en un cluster es un conjunto de particiones y cada partición es un conjunto de registros. Suponiendo que cada partición entra en memoria podemos definir un método llamado "Map Partitions" que aplica una función a cada partición del archivo.

$$\text{MapPartitions}([r]) \rightarrow [r2]$$

Como vemos MapPartitions recibe una lista de registros y genera otra lista de registros de igual o diferente tipo y de igual o diferente cardinalidad. Esta operación es importante para paralelizar tareas en las cuales necesitemos procesar varios registros a la vez. Por ejemplo si buscamos el valor mínimo de un archivo entonces podemos obtener el mínimo de cada partición y finalmente en un reduce obtener el mínimo global.

Si bien Map Partitions es un recurso importante en Map-Reduce en este capítulo vamos a evitar usarlo para ver como podemos resolver problemas usando simplemente el mecanismo basico de Map y Reduce.

3.2.3 Fase de Reduce

La fase de Reduce se encarga de las operaciones que necesiten mas de un registro al mismo tiempo para funcionar. Vamos a definir dos formas de Reduce:

Reduce propiamente dicho

En esta forma reduce funciona realizando operaciones que deben ser asociativas y conmutativas entre registros del archivo. El resultado de un reduce pasa a ser parte del input de la siguiente operación, por ejemplo supongamos que tenemos un archivo formado por números de la forma [1,3,4,-2,5] si hacemos un reduce con la operación "+" lo que haremos es tomar primero "1" y "3" aplicar "+", el resultado nos da "4", a este resultado le aplicamos "+" con -2 y nos da 2, y a este resultado el aplicamos "+" con 5 y nos da el resultado final que es 7. Podemos definir reduce de la forma:

$$\text{Reduce}(r, \text{fun}) \rightarrow \text{val}$$

Es decir que procesamos un conjunto de registros y lo que obtenemos es un valor final. Por ejemplo podemos usar Reduce para encontrar el mínimo de un archivo con una construcción similar a la siguiente:

```
archivo2 = archivo.reduce(lambda x,y: x if x<y else y)
```

Lo que hacemos es ir comparando los elementos de a 2 quedándonos siempre con el mínimo.

Reduce by Key

En esta variante suponemos que cada registro esta formado por una tupla de tipo (clave,valor), notemos que tanto clave como valor pueden ser estructuras de todo tipo: listas, tuplas, diccionarios o valores atómicos. El sistema va a agrupar todos los registros para los cuales la clave es la misma y aplicarles a estos la función reduce. El funcionamiento de reduce es igual al anterior pero el resultado es un par (clave,valor) por cada clave del archivo. Para cada resultado automáticamente se agrega la clave.

$$\text{ReduceByKey}((k, v), \text{fun}) \rightarrow [(k, v)]$$

Para que reduce pueda funcionar el sistema debe asegurarse de juntar todos los registros que tienen igual clave en un mismo equipo, esto implica mover datos en la red y es una operación costosa que se realiza en la fase que llamamos "Shuffle & Sort"

3.2.4 Fase de Shuffle & Sort

La fase de shuffle & sort es la encargada de hacer que el sistema pueda ejecutar la fase de reduce o reduceByKey, en líneas generales implica mover la salida de cada proceso Map a un cierto equipo de forma tal que un proceso "Reducer" pueda procesar todos los registros que le corresponden. Cuando usamos reduceByKey el sistema debe encargarse que todos los registros con la misma clave puedan ser procesados por un mismo proceso Reducer.

La fase de shuffle & sort es siempre la parte mas costosa de todo proceso Map-Reduce por lo que las optimizaciones siempre pasan por minimizar la cantidad de datos que deben moverse de un equipo a otro.

3.3 Normalización

En nuestro primer ejemplo de Map Reduce vamos a normalizar un conjunto de datos de forma tal que todas las columnas (features) tengan promedio cero y desviación standard 1. Este paso es muy importante para muchos algoritmos que funcionan mejor con datos normalizados. Es de destacar que si vamos a normalizar los datos de entrenamiento también tenemos que normalizar los datos de validación y test es decir normalizar todos los datos como si fueran un único set de datos. Cuando no conocemos todos los puntos que queremos testear, clasificar o analizar con nuestro algoritmo la normalización en principio no es posible pero pueden usarse algunos trucos como por ejemplo tomar el logaritmo de cada valor de forma tal de acercarlo un poco a lo que sería una distribución normal. De todas formas dejemos esto para mas adelante y veamos como podemos normalizar un set de datos en Map-Reduce lo cual implica poder calcular el promedio y la desviación standard de cada columna.

El set de datos será el siguiente

```
(1,(1,0,2,3))
(0,(3,1,2,2))
(1,(0,0,-2,4))
(1,(1,1,-1,1))
(0,(2,-2,0,1))
```

El primer elemento de cada vector es su clase, hay solo dos clases 0 y 1. El segundo elemento es el vector propiamente dicho que, como vemos, tiene en nuestro ejemplo cuatro dimensiones. La cantidad de elementos (puntos,vectores) la podemos calcular fácilmente haciendo un count de la cantidad de datos. El count se hace simplemente reemplazando cada dato por "1" y haciendo la suma en un reduce.

Como nos interesa el promedio de cada columna lo que vamos a hacer es convertir a cada vector en una serie de tuplas de tipo (col,valor) indicando el número de columna y su correspondiente valor.

```
kdata = [(1,(1,0,2,3)),(0,(3,1,2,2)),(1,(0,0,-2,4)),(1,(1,1,-1,1)),(0,(2,-2,0,1))]
rsk = sc.parallelize(kdata, 4)
cant = rsk.count()
rsk2 = rsk.flatMap(lambda x:[(idx,val) for idx,val in enumerate(x[1])]))
```

OUTPUT:

```
(0, 1)
(1, 0)
(2, 2)
(3, 3)
(0, 3)
(1, 1)
...etc
```

Una vez que tenemos los datos en este formato es sencillo calcular el promedio simplemente sumando por cada dimensión usando reduceByKey y luego dividiendo por la cantidad total de puntos.

```
rsk2 = rsk.flatMap(lambda x:[(idx,val) for idx,val in enumerate(x[1])])
    .reduceByKey(lambda x,y:x+y)
    .map(lambda x:(x[0],float(x[1])/(cant)))
    .map(lambda x:x[1])
mu = rsk2.collect()

OUTPUT:
[1.4, 0.0, 0.2, 2.2]
```

El siguiente paso es calcular la desviación standard que se calcula de la forma:

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x - \mu)^2}{n - 1}}$$

Es importante recordar que si sabemos el promedio de n valores entonces solo necesitamos $n - 1$ valores para reconstruirlos por eso en la desviación standard se hace el promedio sobre $n - 1$ puntos y no n para compensar el grado de libertad perdido.

El cálculo de la desviación standard en Map-Reduce sigue la misma idea que el promedio, a cada vector le restamos el promedio (μ) y luego generamos registros de tipo (col,valor) en donde el valor lo elevamos al cuadrado. Finalmente acumulamos por dimensión y dividimos por $n - 1$

```
rsk3 = rsk.map(lambda x:tuple(map(sub,x[1],mu)))
    .flatMap(lambda x:[(idx,val**2) for idx,val in enumerate(x)])
    .reduceByKey(lambda x,y:x+y)
    .map(lambda x:(x[1]/(cant-1)))

var = rsk3.map(lambda x:sqrt(x)).collect()

OUTPUT:
[1.140175425099138, 1.224744871391589, 1.7888543819998317, 1.3038404810405297]

Finalmente podemos normalizar cada vector restando el promedio y dividiendo por la desviación standard.

rsknorm = rsk.map(lambda x:(x[0],tuple(map(div,tuple(map(sub,x[1],mu)),var))))
```

OUTPUT:

```
(1, (-0.35082320772281156, 0.0, 1.0062305898749053, 0.6135719910778962))
(0, (1.4032928308912467, 0.8164965809277261, 1.0062305898749053, -0.15339299776947424))
(1, (-1.2278812270298407, 0.0, -1.2298373876248845, 1.3805369799252667))
...etc
```

Normalizar los datos es una parte fundamental en varios algoritmos y por eso es conveniente tener presente como podemos hacerlo usando Map-Reduce para poder procesar datos masivos.

3.4 KNN

Vamos a trabajar ahora con una simple versión de KNN. Vamos a suponer que no hace falta normalizar los datos y usaremos la distancia euclídea. Vamos a mostrar dos variantes en primer lugar el caso mas simple que es $k = 1$ y luego el caso en el cual $k > 1$.

3.4.1 NN

NN es la versión de KNN con $k = 1$ vamos a tener un cierto punto como query o input y vamos a buscar en nuestro set de datos el punto mas cercano al mismo usando alguna métrica como distancia. En este caso vamos a usar la distancia

euclideana pero podría usarse cualquier otra distancia y recordemos que determinar la métrica óptima es muy importante para el correcto funcionamiento de KNN.

El procedimiento es muy simple, calculamos la distancia entre cada punto y nuestro query y generamos un registro de tipo (distancia,(clase,vector)). Luego en la fase de reduce simplemente nos quedamos con el registro que tenga menor distancia comparando de a dos y conservando el menor.

DATOS:

```
(1,(1,0,2,3))
(0,(3,1,2,2))
(1,(0,0,-2,4))
(1,(1,1,-1,1))
(0,(2,-2,0,1))
```

```
q = (0,0,-1,3)
```

```
def euclid(x,y):
    return sqrt(sum((x - y)**2 for x, y in zip(x, y)))

nn = rsknorm.map(lambda x:(euclid(x[1],q),(x[0],x[1])))
    .reduce(lambda x,y:x if x<y else y)
```

OUTPUT:

```
(1.4142135623730951, (1, (0, 0, -2, 4)))
```

Es decir que si estuviéramos clasificando al punto "q" le asignaríamos la clase 1 que es la que corresponde a su vecino mas cercano (0,0,-2,4).

3.4.2 KNN

La extensión lógica a NN es KNN es decir $K > 1$ en este caso el procedimiento es muy similar pero ya no podemos hacer un reduce para quedarnos con el menor punto ya que necesitamos los k vecinos mas cercanos. Vamos a ver un procedimiento que desde el punto de vista teórico no es del todo eficiente pero en la práctica funciona bien. La idea es calcular la distancia a cada punto y luego simplemente ordenar las distancias de menor a mayor y quedarnos los primeros k registros.

```
knn = rsk.map(lambda x:(euclid(x[1],q),(x[0],x[1])))
    .sortByKey().take(3)
```

OUTPUT:

```
(1.4142135623730951, (1, (0, 0, -2, 4)))
(2.449489742783178, (1, (1, 1, -1, 1)))
(3.1622776601683795, (1, (1, 0, 2, 3)))
```

Desde el punto de vista teórico este algoritmo no es eficiente ya que para obtener los k vecinos mas cercanos no necesitamos ordenar todas las distancias que es $O(n \log n)$ sino que podemos simplemente recorrer todas las distancias secuencialmente y quedarnos con las k mas pequeñas y sus correspondientes puntos. Sin embargo el proceso de sort que realiza map-reduce en la fase de shuffle & sort sirve para obtener los primeros k resultados de forma relativamente eficiente por lo que en la práctica esta solución es aceptable.

3.5 Procesamiento de Textos

En este primer ejemplo vamos a suponer que nuestro archivo distribuido está formado por un conjunto de textos (cada registro es un texto), esto puede aplicarse a muchísimas situaciones, por ejemplo un motor de búsqueda, resolución de consultas en lenguaje natural, etc.

3.5.1 Wordcount

En nuestro primer ejemplo lo que queremos devolver es un listado de palabras y frecuencias que nos diga cuantas veces aparece cada palabra en la colección. El primer paso es tomar cada texto del archivo y hacer un "split" generando un registro por cada palabra del mismo.

```
archivo2 = archivo.map(lambda x: x.split())
```

Esto hace algo del estilo:

[cada linea es un registro]

INPUT:

```
hola que tal
que tal todos
hola a todos
```

OUTPUT:

```
hola
que
tal
que
tal
todos
hola
a
todos
```

El siguiente paso es generar por cada palabra un registro de tipo (clave,valor) en donde la clave será la palabra y el valor será 1.

```
archivo3 = archivo2.map(lambda x:(x,1))
```

OUTPUT:
(hola,1)
(que,1)
(tal,1)
(que,1)
(tal,1)
(todos,1)
(hola,1)
(a ,1)
(todos,1)

Una vez que tenemos el archivo en este formato podemos hacer un reduceByKey sumando.

```
archivo4 = archivo3.reduceByKey(lambda x,y:x+y)
```

OUTPUT:
(hola,2)
(que,2)
(tal,2)
(todos,2)
(a ,1)

Y el resultado final es el que queríamos. Eventualmente el sistema puede agregar funciones para ordenar el archivo, tomar los primeros "n" registros a modo de muestra, etc. Toda esta funcionalidad es accesoria, en este capítulo queremos concentrarnos únicamente en como hacer cosas usando únicamente Map y Reduce.

3.5.2 N-gramas

A partir de un texto podemos querer calcular la probabilidad de los diferentes n-gramas ya sea tomando "n" caracteres o "n" palabras. En este ejemplo vamos a usar n-caracteres y lo que queremos calcular es la frecuencia de cada n-grama a lo largo de un conjunto de textos.

Supongamos que queremos usar trigramas (n=3) y que tenemos los siguientes textos:

```
'this is a test'  

'test this thing'  

'this and that'  

'I like this'
```

En primer lugar hay que mapear cada texto a una lista de sus trigramas, esto lo hacemos mediante una función que dado un string devuelve una lista de sus

trigramas y aplicando dicha función a cada texto via flatMap, flatMap deshace la estructura de lista que nos devuelve la función es decir que cada trígrama quedará como un registro en el archivo resultado.

```
def trigrams(t):
    t=t.lower()
    return [t[i:i+3] for i in range(0, len(t) - 2)]
```

```
tdd = tdd.flatMap(lambda x:trigrams(x))
```

OUTPUT:

```
'thi'
'his'
'is '
's i'
' is'
'is '
's a'
...etc..
```

A partir de esta estructura la convertimos en algo de tipo (trígrama,1) y luego hacemos un reduceByKey sumando con lo cual obtenemos la frecuencia de cada trígrama. A modo de ejemplo veamos que podemos ordenar el archivo resultante mediante la función sortBy a la cual le pasamos una función que nos devuelva cuál es la clave de cada registro que usaremos para comparar.

```
tdd = tdd.flatMap(lambda x:trigrams(x))
    .map(lambda x:(x,1))
    .reduceByKey(lambda x,y:x+y)
    .sortBy(ascending=False,keyfunc=lambda x:x[1])
```

OUTPUT:

```
('thi', 5)
('his', 4)
(' th', 4)
('is ', 4)
('s a', 2)
('tes', 2)
etc...
```

A partir de las frecuencias es posible acumular todas las frecuencias para calcular el total y luego dividir cada frecuencia por este total para obtener la probabilidad de cada trígrama.

```
total = tdd.map(lambda x:x[1]).reduce(lambda x,y:x+y)
tddr = tdd.map(lambda x:(x[0],round(float(x[1])/total,3)))
```

```
OUTPUT:
('thi', 0.111)
('his', 0.089)
(' th', 0.089)
('is ', 0.089)
('s a', 0.044)
('tes', 0.044)
('est', 0.044)
etc...
```

Un ejercicio similar es devolver por cada trígrama la lista de caracteres que siguen al trígrama y su probabilidad para hacer esto hay que generar registros de tipo (trígrama, carácter) y luego acumular por clave (trígrama) calculando la probabilidad de cada carácter que sigue al trígrama.

3.6 Join

En muchos casos es necesario trabajar con más de un archivo distribuido al mismo tiempo y realizar un "join" entre ambos. En un Join lo que hacemos es generar un nuevo archivo en donde los registros de este nuevo archivo surgen de combinar, unir un registro del archivo "A" y otro del archivo "B". Según la condición del join se decide cuando dos registros de los archivos conforman un nuevo registro del archivo resultante. La opción más simple es combinar todos los registros de "A" con todos los registros de "B", a esta operación la llamamos producto cartesiano.

3.6.1 Producto Cartesiano

Supongamos los siguientes archivos: el primero tiene información sobre alumnos en formato (padrón, nombre y apellido) y el segundo tiene información sobre materias aprobadas en formato (padrón, materia, nota).

```
Archivo "A":
(71023, "Juan Perez")
(72093, "Jorge Gomez")
```

```
Archivo "B"
(71023, 7506, 7)
(72093, 7509, 4)
(72093, 7506, 6)
(72093, 7507, 7)
```

El resultado del producto cartesiano surge de combinar a cada registro de "A" con cada registro de "B" como tenemos dos y cuatro registros respectivamente el producto cartesiano tendrá un total de ocho registros.

A X B:

```
(71023, "Juan Perez", 71023, 7506, 7)
(71023, "Juan Perez", 72093, 7509, 4)
(71023, "Juan Perez", 72093, 7506, 6)
(71023, "Juan Perez", 72093, 7507, 7)
(72093, "Jorge Gomez", 71023, 7506, 7)
(72093, "Jorge Gomez", 72093, 7509, 4)
(72093, "Jorge Gomez", 72093, 7506, 6)
(72093, "Jorge Gomez", 72093, 7507, 7)
```

Algunos de estos nuevos registros no tienen mucho sentido desde el punto de vista de la semántica de los datos ya que combinan los datos de un alumno con la nota de una materia aprobada por otro alumno. Para que el resultado tenga sentido (en este caso) sería conveniente filtrar la salida de forma tal que nos quedemos únicamente con aquellos registros en los cuales el número de padrón es el mismo para ambos registros. Cuando en el resultado final incluimos únicamente las combinaciones de registros para las cuales una cierta clave/campo coincide en ambos estamos en presencia de un "Inner Join".

3.6.2 Inner Join y Outer Join

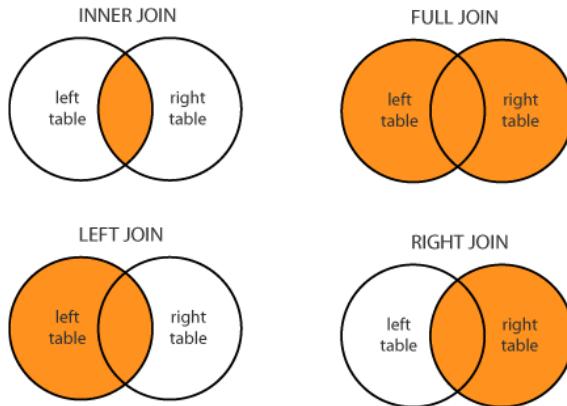


Figure 3.2: Los distintos tipos de Join.

El producto cartesiano es decir "todos contra todos" se puede llamar también "full join". Cuando queremos incluir todos los registros del archivo "A" y los registros del archivo "B" cuya clave coincide con "A" tenemos un "left join" un ejemplo de "left join" es el siguiente considerando "id" como campo clave:

Archivo "A" (id,valor):

```
(1,15)
(2,24)
(3,31)
Archivo "B" (id,valor):
(2,44)
(3,53)
(4,13)
```

```
A LEFT JOIN B
(1,15,NA,NA)
(2,24,2,44)
(3,31,3,53)
```

Como puede verse todos los registros de "A" están presentes ya sea que tengan un match con algún registro de "B" o no.

De la misma forma un "right join" deja en los resultados todos los registros de B independientemente de que haya un match con el archivo "A".

Los joins son operaciones relativamente costosas en Map Reduce que involucran la necesidad de la fase de Shuffle & Sort pero en muchos procesos son imprescindibles al ser la única forma de combinar el resultado de dos archivos distribuidos.

3.7 Operaciones con Matrices

En esta sección vamos a operar con matrices dispersas almacenada en un archivo distribuido, cada registro es de la forma: (fila,columna,valor)

Por ejemplo el siguiente archivo:

```
(1,2,4)
(1,5,3)
(2,1,3)
(3,2,2)
(4,4,-1)
(5,1,1)
(5,5,2)
```

Corresponde a la siguiente matriz:

$$\begin{pmatrix} 0 & 4 & 0 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 2 \end{pmatrix}$$

3.7.1 Multiplicación matriz-vector

Nuestro objetivo es calcular el producto entre una matriz y un vector, supongamos que el vector entra en memoria y está almacenado en formato de una lista de tantos elementos como columnas tiene la matriz. En este caso como la matriz es de 5x5 el vector será de 5x1.

Supongamos que nuestro vector es [1,2,3,4,5]:

$$\begin{pmatrix} 0 & 4 & 0 & 0 & 3 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} =$$

Para resolver el producto tenemos que multiplicar cada fila de nuestra matriz por el vector. Esto es equivalente a multiplicar cada elemento de la matriz dispersa por el elemento que le corresponde en el vector y acumular por filas. Vamos a realizar toda la operación paso a paso para poder ver como funciona esto usando Map reduce.

La matriz dispersa viene en formato de triples (fila, columna, valor) queremos que la fila sea la clave es decir convertir (fila,columna,valor) en (fila,(columna,valor)). Esto lo logramos con una sencilla operación Map.

```
m2 = matrixRDD.map(lambda x:(x[0],(x[1],x[2])))
```

OUTPUT:

```
(1, (2,4))
(1, (5,3))
(2, (1,3))
(3, (2,2))
(4, (4,-1))
(5, (1,1))
(5, (5,2))
```

Notemos que en la mayoría de los lenguajes Map Reduce las tuplas y listas se indexan desde cero, nuestra matriz dispersa la hemos indexado a partir de 1, esto es algo a tener en cuenta.

Una vez que tenemos la matriz dispersa en un formato que nos conviene podemos multiplicar cada elemento de la matriz por el elemento del vector que le corresponde. Cada elemento de la matriz nos indica su número de columna y ese número de columna menos 1 nos indica el índice del vector por el que tenemos que multiplicar. Esto lo hacemos, nuevamente usando una operación Map.

```
m3 = m2.map(lambda x:(x[0],(vector[x[1][0]-1]*x[1][1])))
```

OUTPUT:

```
(1,8)
(1,15)
(2,3)
(3,4)
(4,-4)
(5,1)
(5,10)
```

El último paso es muy sencillo ya que solo tenemos que hacer un reduceByKey sumando los elementos:

```
m4 = m3.reduceByKey(lambda x,y:(x+y))
```

OUTPUT:

```
(1,23)
(2,3)
(3,4)
(4,-4)
(5,11)
```

Notemos que el resultado es un vector en formato (fila,valor).

Este proceso ha ilustrado el uso de Map y Reduce para manipular la forma de cada registro de forma tal de poder realizar las operaciones necesarias entre los mismos y dejarlos listos para que un reducer pueda generarnos el resultado final.

3.7.2 Multiplicación matriz-matriz

Ahora que hemos visto en detalle la multiplicación de una matriz por un vector usando Map-Reduce y la operación de join podemos ver como realizar el producto entre dos matrices dispersas, suponiendo que sus dimensiones sean compatibles. A efectos de que el ejemplo sea lo mas claro y simple posible vamos a realizar la operación con las siguientes matrices de 2x2

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} =$$

Para realizar el producto entre dos matrices debemos notar que los elementos de la primer columna de la matriz 1 (1 y 3) siempre se multiplican únicamente por los elementos de la primera fila de la matriz 2 (5 y 6) por lo tanto la estrategia es realizar un join en donde la columna de la matriz 1 coincida con la fila de la matriz 2. Las matrices las tenemos almacenadas en el siguiente formato:

M1:

```
(1,1,1)
(1,2,2)
(2,1,3)
(2,2,4)
```

```
M2:  
(1,1,5)  
(1,2,6)  
(2,1,7)  
(2,2,8)
```

Vamos a usar una operación Map para convertir cada matriz en el formato (clave, valor) en donde para la matriz 1 la clave será el número de columna y para la matriz 2 la clave será el número de fila.

```
r1 = m1.map(lambda x:(x[1],(x[0],x[2])))  
r2 = m2.map(lambda x:(x[0],(x[1],x[2])))
```

OUTPUT:

```
R1:  
(1,(1,1))  
(1,(2,3))  
(2,(1,2))  
(2,(2,4))
```

```
R2:  
(1,(1,5))  
(1,(2,6))  
(2,(1,7))  
(2,(2,8))
```

Ahora podemos realizar un join entre r1 y r2:

```
rz = r1.join(r2)
```

OUTPUT:

```
(1, ((1, 1), (1, 5)))  
(1, ((1, 1), (2, 6)))  
(1, ((2, 3), (1, 5)))  
(1, ((2, 3), (2, 6)))  
(2, ((1, 2), (1, 7)))  
(2, ((1, 2), (2, 8)))  
(2, ((2, 4), (1, 7)))  
(2, ((2, 4), (2, 8)))
```

Observemos ahora que en este archivo tenemos toda la información necesaria para realizar el producto entre las matrices, tenemos que multiplicar los valores y el resultado hay que acumularlo en el número de fila y columna indicado en los registros. Por ejemplo para el registro (2, ((2, 4), (1, 7))) tenemos que multiplicar $4 * 7$ y ese resultado acumularlo para la fila2, columna1. Vamos entonces a realizar los productos mediante Map dejando como clave (fila, columna)

para luego poder acumular (sumar) mediante un reduce. Hay que tener mucho cuidado de usar los índices correctamente por favor prestar atención y asegurarse que entienden como funciona la siguiente línea.

```
rj2 = rj.map(lambda x:((x[1][0][0],x[1][1][0]),x[1][0][1]*x[1][1][1]))  
  
OUTPUT:  
((1, 1), 5)  
((1, 2), 6)  
((2, 1), 15)  
((2, 2), 18)  
((1, 1), 14)  
((1, 2), 16)  
((2, 1), 28)  
((2, 2), 32)
```

Finalmente solo nos queda acumular mediante reduceByKey

```
rj3 = rj2.reduceByKey(lambda x,y:x+y)  
  
OUTPUT:  
((1, 2), 22)  
((1, 1), 19)  
((2, 2), 50)  
((2, 1), 43)
```

Y el resultado es el producto de M1*M2

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Notar que la misma aproximación funcionaría para multiplicar una matriz por un vector cuando ninguno de los dos entra en memoria, es igual a multiplicar dos matrices solo que m2 tiene una sola columna. De esta forma podríamos realizar productos entre matrices o entre una matriz y un vector que sea dispersas y que tengan millones o incluso billones de registros paraleizando la operación dentro de un cluster. Esto es muy importante ya que muchos algoritmos importantes se basan en realizar operaciones entre matrices.

3.8 Redes Sociales

En esta sección vamos a analizar el procesamiento de redes sociales usando Map Reduce. En un formato muy básico podemos definir las relaciones de una red social mediante registros de tipo (A,B) indicando que "A" es amigo de "B" o bien que "A" sigue a "B".

3.8.1 Relaciones no correspondidas

Como primer ejercicio supongamos que (A,B) indica que "A" sigue a "B" y queremos encontrar todas las relaciones no correspondidas es decir que existe (A,B) pero no existe (B,A), supongamos el siguiente set de datos:

```
rs=
('A','B')
('A','C')
('B','C')
('B','A')
('C','B')
```

Para resolver este problema vamos a tomar cada registro de nuestra red social y lo vamos a ordenar, de esta forma (A,B) y (B,A) quedan convertidos en (A,B). Luego podemos simplemente generar registros de tipo ((A,B),1) y haciendo un ReduceByKey obtener los registros que sumen 1 que son aquellas amistades no correspondidas. En Spark el proceso completo es el siguiente:

```
rs = [ ('A','B'), ('A','C'), ('B','C'), ('B','A'), ('C','B') ]
rsRDD = sc.parallelize(rs, 4)
print rsRDD.map(lambda x:tuple(sorted(x)))
    .map(lambda x:(x,1))
    .reduceByKey(lambda x,y:x+y)
    .filter(lambda x:x[1]==1)
    .collect()
```

3.8.2 Contando Triángulos en una Red Social

Una de las características mas importantes de una Red Social es que tienen una cantidad inusual de triángulos, es decir tienen una cantidad de triángulos que es muy superior a la cantidad que tendría una red aleatoria con igual cantidad de nodos y aristas. Esto se debe a que cuando "A" es amigo de "B" y "C" es muy probable que "B" y "C" se relacionen cerrando el triángulo, el motivo es que ambos conocen un nodo en común: "A".

En este ejemplo vamos a usar Map-Reduce para contar la cantidad de triángulos en una red social a partir de un archivo de tipo (A,B) indicando que "A" y "B" son amigos.

El archivo que vamos a usar como ejemplo es el siguiente:

```
('A','B')
('B','C')
('C','D')
('A','C')
('D','B')
('D','E')
('E','B')
('B','D')
```

La estrategia que vamos a usar tiene dos partes. En primer lugar vamos a determinar todas las relaciones de tipo $A \rightarrow B \rightarrow C$ luego usaremos estas relaciones para verificar si existe la relación $A \rightarrow C$ o $C \rightarrow A$ que cierre el triángulo. Suponemos en este ejercicio que las relaciones de amistad son bilaterales es decir que si "A" es amigo de "B" entonces "B" es amigo de "A" pero en el archivo guardamos solo (A,B) o (B,A) indistintamente. Un buen comienzo es entonces ordenar cada registro del archivo para que siempre quede primero el usuario "menor".

```
rs2RDD = rs2RDD.map(lambda x:tuple(sorted(x)))
```

OUTPUT:

```
('A', 'B')
('B', 'C')
('C', 'D')
('A', 'C')
('B', 'D')
('D', 'E')
('B', 'E')
('B', 'D')
```

Ahora vamos a buscar todas las triples (A,B,C) tales que exista (A,B) y (B,C). Para hacer esto vamos a generar por cada registro de tipo (A,B) dos registros uno de tipo (A,(B,0)) y otro (B,(A,1)) lo que indicamos es que para la clave "A" encontramos "B" a su derecha (0) y para la clave "B" encontramos "A" a su izquierda (1).

```
res3= rs2RDD.flatMap(lambda x:[(x[1],(x[0],1)),(x[0],(x[1],0))])
```

```
('B', ('A', 1))
('A', ('B', 0))
('C', ('B', 1))
('B', ('C', 0))
('D', ('C', 1))
('C', ('D', 0))
('C', ('A', 1))
('A', ('C', 0))
('D', ('B', 1))
('B', ('D', 0))
('E', ('D', 1))
('D', ('E', 0))
('E', ('B', 1))
('B', ('E', 0))
('D', ('B', 1))
('B', ('D', 0))
```

Podemos ahora agrupar por clave (groupByKey) que es lo mismo que un

reducir que no hace nada con los registros salvo acumularlos en una lista. Una vez que agrupamos por clave tendremos por ejemplo algo del estilo:

```
('B', [('A', 1), ('C', 0), ('D', 0)])
```

Indicando que para el usuario "B" hemos observado a "A" a su izquierda y "C", "D" a su derecha. Es decir que existen (A,B) y (B,C),(B,D). Lo que vamos a hacer entonces es generar (A,B,C) y (A,B,D) es decir que vamos a hacer las combinaciones de todos los que aparecen con 0 contra todos los que aparecen con 1 poniendo a la clave en el medio. Esto lo vamos a hacer mediante una función.

```
def proc(key,vals):
    print key
    l0 = [node[0] for node in vals if node[1]==1]
    l1 = [node[0] for node in vals if node[1]==0]
    join = list(itertools.product(l0,l1))
    res = [(node[0],key,node[1]) for node in join if len(node[0])>0 and len(node[1])>0]
    return res
```

```
res3= res3.groupByKey().flatMap(lambda x:proc(x[0],x[1])).distinct()
```

OUTPUT:

```
('A', 'B', 'E')
('C', 'D', 'E')
('A', 'B', 'C')
('A', 'C', 'D')
('B', 'C', 'D')
('B', 'D', 'E')
('A', 'B', 'D')
```

La siguiente etapa es mucho mas simple, como sabemos que existe (A,B,E) lo que haremos es generar un registro con clave (A,E) y valor (A,B,E) que quiere decir que si existe (A,E) entonces se forma el triángulo (A,B,E). A este archivo luego le hacemos un join con el archivo original en donde convertimos (A,B) en algo de tipo ((A,B),1) es decir clave (A,B) y valor 1. Como ejemplo pensemos que tenemos el registro (A,B,E) de la salida anterior, lo convertimos en ((A,E),(A,B,E)) es decir clave (A,E) y valor (A,B,E) por otro lado tenemos el registro (A,E) del archivo original que convertimos en ((A,E),1) cuando hacemos el join nos queda algo de tipo ((A,E),((A,B,E),1)). Notar que para los triángulos que no se cierran no se genera nada ya que si (A,E) no existe en el archivo original no se genera resultado alguno con clave (A,E) en el archivo final por tratarse de un inner join. Finalmente de los registros formamos extraemos el primer componente del segundo componente de la tupla para tener el triángulo. Si solo nos interesa la cantidad de triángulos simplemente hacemos un count, o lo que es igual a convertir cada registro en un 1 y sumar mediante un reduce.

```
rs2k = rs2RDD.map(lambda x:(x,1))
res4 = res3.flatMap(lambda x:[((x[0],x[2]),x),((x[2],x[0]),x)])
res5 = res4.join(rs2k).map(lambda x:tuple(sorted(x[1][0]))).distinct()

OUTPUT:
('A', 'B', 'C')
('B', 'D', 'E')
('B', 'C', 'D')
```

Chapter 4

Complejidad, Compresión y Teoría de la Información

Information is the resolution of uncertainty.

Claude Shannon

En este capítulo vamos a intentar entender los datos mediante la complejidad de los mismos, la complejidad está directamente relacionada con la cantidad de bits necesarios para almacenar los datos y por lo tanto es un tema íntimamente ligado a la compresión de datos. Para poder comprimir bien es necesario poder modelar los datos de la mejor forma posible y para poder modelar los datos de la mejor forma posible es necesario entender los datos. Por lo tanto podemos decir que comprimir y entender son sinónimos. Aprender técnicas de compresión de datos nos permitirá entonces poder entender a los datos de la mejor forma posible lo cual es algo sumamente importante. Veremos en este capítulo como la compresión de datos puede usarse no solo para que los datos ocupen menos espacio sino también como una forma de representar el conocimiento que tenemos sobre los mismos.

4.1 Complejidad

Hay varias formas de definir complejidad. En este capítulo vamos a usar la complejidad de Kolmogorov que define la complejidad de un cierto string/archivo o dato como la cantidad de bits que ocupa el programa mínimo que puede generar dicho string.

4.1.1 Complejidad de Kolmogorov

Definition 4.1 (Complejidad de Kolmogorov) *Sea x un string, $K(x)$ es igual a la cantidad de bits mínimos que debe tener un programa que genera x*

Veamos algunos ejemplos para entender el concepto de complejidad:

```
x1 = 000000000000000000000000
x2 = 87490423166544426710
x3 = 94482553797747268471
```

El primer string es una serie de veinte ceros, por lo tanto podríamos escribir un programa muy cortito de la forma *for i=1 to 20 print "0"*, si el string tuviera miles de millones de ceros el programa crecería muy poco en tamaño, apenas lo necesario para guardar la cantidad de ceros a emitir. Por lo tanto podemos razonar que este string tiene complejidad cero. El segundo string es aleatorio, por lo tanto para que un programa lo genere debería simplemente hacer *print "87490423166544426710"* la longitud del programa es entonces, en el límite igual a la cantidad de bits necesarios para representar el string y la complejidad del string es la máxima posible. El tercer string es mas interesante, a primera vista podría parecer random pero sin embargo no lo es, son dígitos de π a partir de un offset pequeño. El programa necesario para generar este string es entonces muy pequeño ya que existe código extremadamente mínimo para generar dígitos de π y solo nos haría falta codificar el offset que es pequeño. Por lo tanto la complejidad de este string es muy baja.

Nuestro ejemplo sirve para presentar varios conceptos muy importantes: en primer lugar que puede ser muy difícil darnos cuenta de la complejidad de un cierto conjunto de datos, un programa debería darse cuenta que nuestro archivo "x2" contiene dígitos de π para encontrar la forma mínima de representarlo. El lenguaje de programación a usar, compilador y demás influyen en la longitud del programa mínimo que estamos buscando pero en general es algo que podemos ignorar ya que a medida que los strings son lo suficientemente grandes el overhead introducido por el programa tiende a cero.

La complejidad de Kolmogorov nos sirve para poder definir de forma precisa un concepto bastante complejo: el azar.

Definition 4.2 (Datos Aleatorios) *Sea x un string, decimos que x es random/aleatorio si y sólo si $K(x) = |x|$. Es decir que la complejidad del string es igual a la longitud del mismo.*

4.1.2 Propiedades de $K(x)$

Vamos a ver ahora algunas propiedades de la complejidad de Kolmogorov y a explicar sus fundamentos. xy denota la concatenación de los strings x e y .

1. $K(x) \geq 0$
2. $K(x) \leq |x|$
3. $K(xy) \leq K(x) + K(y)$
4. $K(xy) \geq K(x), K(y)$

5. $K(xy) = K(yx)$
6. $K(xx) = K(x)$
7. $K(xy) + K(z) \leq K(xz) + K(yz)$

Las dos primeras propiedades son simples: la complejidad no puede ser negativa (es el tamaño en bits de un programa) y no puede ser mayor a la longitud del dato ya que de lo contrario podríamos simplemente hacer *print x* para generarlo. Recordemos que el overhead del lenguaje de programación es despreciable.

La propiedad 3 dice que la complejidad de la concatenación de dos strings nunca puede ser mayor a la suma de la complejidad de los mismos. Esto se demuestra de la misma forma que la propiedad 2.

La cuarta propiedad dice que la complejidad de concatenar dos strings es mayor a la complejidad de cada uno de ellos. Esto se puede demostrar por el absurdo. Si $K(x + y) < K(x)$ entonces podríamos representar a "x" con el programa que genera x+y y simplemente descartar "y".

La quinta propiedad es la simétrica y establece que la complejidad de dos strings concatenados no se ve afectada por el orden en el cual concatenamos los mismos.

La sexta propiedad es muy importante y establece que la complejidad de un string concatenado consigo mismo es igual a la complejidad del string. Esto es muy fácil de demostrar ya que si tenemos un programa que genera x entonces simplemente haciendo un ciclo for obtenemos "n" veces x sin agregar longitud al programa.

Finalmente la propiedad número siete es la ley distributiva. Y es fundamental para explicar algunas otras propiedades por lo que vamos a demostrarla.

$$K(xy) + K(z) \leq K(xz) + K(yz)$$

Como sabemos que $K(xy) \leq K(x) + K(y)$ podemos descomponer la parte derecha de la desigualdad sin afectar la misma.

$$K(xy) + K(z) \leq K(x) + K(z) + K(y) + K(z)$$

$$K(xy) \leq K(x) + K(y) + K(z)$$

Y esto es válido ya que sabemos que $K(xy) \leq K(x) + K(y)$ por lo tanto también será menor si agregamos $K(z)$ o cualquier otra constante.

Mediante la complejidad de Kolmogorov es posible estimar la distancia entre un string y otro que llamaremos distancia de Kolmogorov.

Definition 4.3 (Distancia de Kolmogorov)

$$KD(x, y) = K(xy) - \min\{K(x), K(y)\}$$

Notemos que la distancia de Kolmogorov nos da un número que va desde 0 cuando $K(xy) = \min\{K(x), K(y)\}$ hasta $\max\{K(x), K(y)\}$ por lo tanto si queremos un número entre 0 y 1 podemos dividir por el máximo entre $K(x)$ y $K(y)$ para obtener un valor entre 0 y 1. Esta es la distancia normalizada de Kolmogorov.

Definition 4.4 (Distancia Normalizada de Kolmogorov)

$$NKD(x, y) = \frac{K(xy) - \min\{K(x), K(y)\}}{\max\{K(x), K(y)\}}$$

Definimos la distancia entre dos strings como la complejidad de los strings concatenados menos la complejidad del string de menor complejidad entre x e y. Esto es equivalente a decir que si tenemos un programa que genera x o y cuantos bits tenemos que agregarle para generar la concatenación de x e y. Veamos si esta distancia sirve como métrica validando que cumpla las tres propiedades que sabemos debe cumplir una distancia:

1. $KD(x, y) \geq 0$
2. $KD(x, y) = KD(y, x)$
3. $KD(x, y) \leq KD(x, z) + KD(z, y)$

La primera propiedad es fácil de demostrar ya que $K(xy)$ es mayor a $K(x)$ y $K(y)$ por lo tanto es mayor al mínimo entre ambos y por lo tanto la distancia de Kolmogorov siempre es positiva.

Para la segunda propiedad sabemos que $K(x + y) = K(y + x)$ por lo tanto :

$$KD(x, y) = K(yx) - \min\{K(x), K(y)\} = KD(y, x)$$

La tercer propiedad es la desigualdad triangular

$$KD(x, y) \leq KD(x, z) + KD(z, y)$$

$$K(xy) - \min\{K(x), K(y)\} \leq K(x+z) - \min\{K(x), K(z)\} + K(z+y) - \min\{K(z), K(y)\}$$

Supongamos $K(x) \leq K(y) \leq K(z)$ sin perder generalidad.

$$K(xy) - K(x) \leq K(xz) - K(x) + K(zy) - K(y)$$

Sumando $K(z)$ a ambos lados

$$K(xy) + K(z) \leq K(xz) + K(zy) - K(y) + K(z)$$

Por la propiedad distributiva sabemos que $K(xy) + K(z) \leq K(xz) + K(zy)$ y como $K(z) > K(y)$ a la parte derecha le estamos agregando una constante positiva y vale la desigualdad

$$K(xy) \leq K(xz) + K(zy) + \lambda^+$$

Con lo cual demostramos la desigualdad triangular.

4.1.3 Intractabilidad de $K(x)$

Hemos visto la utilidad de la complejidad de Kolmogorov para calcular la complejidad de un string y las propiedades de la misma. Sin embargo vamos a demostrar que $K(x)$ es intractable o incomputable, es decir no existe forma de calcular $K(x)$

Theorem 4 (Kolmogorov) $K(x)$ es intractable.

Vamos a escribir un programa que vaya generando todos los strings posibles en binario (0,1,10,11,100,101,...) hasta encontrar el primero cuya complejidad es de 200Mb. Cuando lo encontramos simplemente lo imprimimos y salimos del programa.

```
for i=1 to Inf (cantidad de bits)
    for j=0 to 2^i-1
        if K(j)>=200Mb then EXIT
    end
end
```

Notemos ahora que el programa que hemos escrito ocupa, evidentemente, menos de 200Mb por lo tanto hemos escrito un programa de menos de 200Mb que genera un string cuya complejidad es mayor a 200Mb y esto genera un absurdo. Es decir que hemos demostrado que $K(x)$ no puede existir ya que si existiera entonces nuestro programa invalidaría su definición. Esta demostración necesita un cierto tiempo de maduración así que invitamos al lector a realizar una pausa y, contemplativamente, pensar en lo que acabamos de demostrar.

Es desafortunado demostrar que $K(x)$ no existe pero es posible estimar $K(x)$ mediante un compresor de datos. Como $K(x)$ es la longitud en bits del programa mas chico que genera x podemos aproximar $K(x)$ por el mejor resultado posible de aplicar un compresor de datos a x es decir $C(x)$. De esta forma el mejor compresor posible existente hasta la fecha nos determina la complejidad del string. Todas las propiedades de $K(x)$ se mantienen con algunos cuidados especiales para $C(x)$ y podemos entonces definir la distancia normalizada de compresión.

4.1.4 Distancia Normalizada de Compresión (NCD)

Definition 4.5 (Distancia Normalizada de Compresión)

$$NCD(x, y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}$$

Esta distancia es muy interesante ya que nos permite calcular la distancia entre dos archivos o dos strings sin absolutamente ningún tipo de parámetro, por eso decimos que es una distancia no-paramétrica. Podemos usar esta distancia para KNN o cualquier otro algoritmo que necesite de una distancia. Para que la NCD sea efectiva $C(x)$ debe cumplir las propiedades de $K(x)$ y para eso

necesitamos establecer algunos requisitos sobre el compresor de datos a utilizar: principalmente el compresor no debe trabajar por bloques, sino procesando el archivo como una totalidad, si el compresor trabajara por bloques entonces el resultado de comprimir la concatenación de un archivo con otro no tendría el mismo efecto ya que dependiendo del tamaño del bloque lo aprendido sobre el primer archivo podría no usarse para comprimir el segundo y es clave que esto ocurra para que la NCD funcione correctamente como distancia.

Notemos que valen las siguientes propiedades para $C(x)$ si usamos un compresor adecuado:

1. $C(xy) \geq C(x)$
2. $C(xy) = C(yx)$
3. $C(xx) = C(x)$

4.2 Teoría de la Información

La teoría de la información es una rama de la computación creada por Claude Shannon en su tratado "A Mathematical Theory of Communication" [Shannon]. En este trabajo Shannon estudió las propiedades de la información almacenada en forma digital en formato binario y estableció toda la teoría necesaria para poder comprimir dicha información y transmitirla a través de un canal ruidoso con la posibilidad de recuperar la información al otro lado. Este trabajo es la piedra fundamental de la teoría de compresión y comunicación de datos. En nuestro curso vamos a usar el concepto de entropía definido por Shannon como una forma basada en probabilidades de medir la complejidad de un string o archivo es decir otra forma de aproximarnos a $K(x)$.

Veamos algunas definiciones necesarias para llegar al concepto de entropía.

4.2.1 Mensajes y Alfabetos

Un alfabeto es un conjunto de símbolos distintos. Por ejemplo $\{0,1\}$ es el alfabeto binario que será el que vamos a usar universalmente en este apunte.

Un mensaje es un conjunto de símbolos de un determinado alfabeto. Por ejemplo "A" es un mensaje de 8 bits, usando 8 símbolos del alfabeto binario. En este apunte usaremos mensajes de 8 bits, aunque podrían ser 16 si usamos Unicode.

4.2.2 Fuente

Una fuente es un conjunto de mensajes. En nuestro caso las fuentes serán archivos o strings, es decir un conjunto de caracteres de 8 bits cada uno. Es de destacar que esto abarca cualquier archivo posible independientemente de su significado o contenido.

4.2.3 Códigos

Un código es un conjunto de símbolos de un determinado alfabeto usado para representar un mensaje. En general los mensajes tienen longitud fija pero los códigos pueden tener longitud fija o variable. Existe una función de codificación que transforma un mensaje en su correspondiente código.

$$f(m_i) = c_i$$

Vamos a definir que un código es decodificable si y sólo si para cualquier conjunto de códigos solo existe un único conjunto de mensajes capaz de generar dichos códigos.

char	Cod1	Cod2	Cod3
A	0	10	0
B	010	00	10
C	01	11	110
D	10	110	111

Table 4.1: Ejemplo códigos.

En la tabla tenemos tres códigos diferentes para representar los caracteres "A", "B", "C", "D". Tenemos que analizar si dichos códigos son decodificables. Para demostrar que un código no es decodificable tenemos que encontrar una secuencia de códigos ambigua, es decir que pueda corresponder a mas de un conjunto de mensajes. Podemos ver que el primer código es ambiguo ya que el código "010" puede corresponder a "B" o a "AD" por lo tanto no es un código decodificable. Los otros dos códigos son decodificables y es difícil demostrar que lo sean, por el momento digamos que no es posible encontrar una combinación de códigos que resulten ambiguos. El test de Sardinas y Patterson puede usarse para comprobar si un código es decodificable pero no es tema de nuestro apunte.

Códigos prefijos

Existe un conjunto de códigos que siempre son decodificables llamados códigos prefijos. La definición de código prefijo es la siguiente:

Definition 4.6 (Código Prefijo) *Decimos que un código es prefijo si no existe ningún código que tenga un prefijo igual a otro código completo.*

Es decir que si tenemos por ejemplo el código "010" entonces ningún otro código puede comenzar con "010". En nuestro ejemplo anterior el código 3 es prefijo, el código 2 no lo es ya que el código de "D" comienza con 11 que es el código de "C".

Theorem 5 (Códigos Prefijos) *Si un código es prefijo entonces es decodificable.*

Observemos que la implicación no vale en la otra dirección. Un código puede ser decodificable y no ser prefijo. Los códigos prefijos se llaman también instantáneos ya que pueden decodificarse unívocamente recorriéndolos de izquierda a derecha.

Los códigos decodificables cumplen con la llamada "desigualdad de Kraft"

Desigualdad de Kraft

Definition 4.7 (Desigualdad de Kraft) *Sea l_i la longitud de cada código*

$$\sum_i 2^{-l_i} \leq 1$$

Theorem 6 (Kraft) *Para cualquier conjunto de longitudes que cumplan la desigualdad de Kraft existe un código prefijo con códigos de dicha longitud.*

Veamos que ocurre para los códigos que vimos en nuestro ejemplo. El código 1 tiene longitudes $(1,3,2,2)$ y $\sum_i 2^{-l_i} = 1.125$ el código no cumple la desigualdad de Kraft y por lo tanto no es decodificable. Para el código 2 las longitudes son $(2,2,2,3)$ y la sumatoria nos da 0.825 si bien el código que mostramos no era prefijo sabemos que existe un código prefijo con longitudes $(2,2,2,3)$. Finalmente el tercer código tiene sumatoria = 1 y es prefijo.

4.2.4 Entropía

La entropía de una fuente se define como la esperanza (valor esperado) de la longitud de sus códigos. Cada mensaje diferente m_i tiene una probabilidad de ocurrir P_i y un código asignado c_i de longitud l_i la entropía se calcula entonces como:

$$H(X) = \sum_{i=1}^n P_i l_i$$

La entropía puede entenderse de diferentes formas. Por un lado es una longitud promedio, es decir la longitud promedio en bits de los códigos usados para representar un mensaje. Es también una medida de la cantidad de información representada en la fuente. Si queremos representar a una fuente en la menor longitud posible tenemos que minimizar su entropía.

Planteamos entonces:

$$(MIN) \sum_i P_i l_i$$

Y sabemos además que el código tiene que cumplir con la desigualdad de Kraft:

$$\sum_i 2^{-l_i} \leq 1$$

Aplicando multiplicadores de Lagrange podemos escribir el siguiente Lagrangiano:

$$\mathcal{L} = \sum P_i l_i + \lambda (\sum 2^{-l_i})$$

Diferenciando respecto de l_i obtenemos:

$$\frac{\partial \mathcal{L}}{\partial l_i} = P_i - \lambda 2^{-l_i} \log_e 2$$

Igualando a cero:

$$2^{-l_i} = \frac{P_i}{\lambda \log_e 2}$$

Reemplazando esta ecuación en la restricción para encontrar λ tenemos:

$$\lambda = \frac{1}{\log_e 2}$$

Y por lo tanto:

$$P_i = 2^{-l_i}$$

Entonces:

$$l_i = -\log_2 P_i$$

De esta forma demostramos que para un mensaje con probabilidad P_i el código de menor longitud (óptimo) tiene longitud igual a $-\log_2 P_i$.

Podemos ahora reemplazar la fórmula del l_i óptimo en la fórmula de la entropía para obtener:

Definition 4.8 (Entropía)

$$H(X) = - \sum_{i=1}^n P_i \log_2 P_i$$

Es decir que hacemos un promedio ponderado por las probabilidades de las longitudes óptimas de los códigos $\log_2(P(x_i))$.

El concepto de entropía es fundamental en este curso y vamos a desarrollarlo bastante por lo que es conveniente ver algunos ejemplos y notación para poder tener el concepto afianzado.

Calcular la entropía del siguiente string.

ABAABC

Notemos que para calcular la entropía solo necesitamos la probabilidad de cada uno de los caracteres diferentes (mensajes) del archivo/string.

$$A = 3/6$$

$$B = 2/6$$

$$C = 1/6$$

Como para el cálculo de la entropía solo interesan las probabilidades de cada mensaje vamos a usar la siguiente notación: $H[3/6; 2/6; 1/6]$ Calculando:

$$H = -[\frac{3}{6} \log_2 \frac{3}{6} + \frac{2}{6} \log_2 \frac{2}{6} + \frac{1}{6} \log_2 16] = 1.4591$$

Ahora que sabemos calcular la entropía tenemos que analizar su significado. Hay dos conceptos fundamentales: la entropía se mide en **bits** y es un **promedio**. Es decir que nos da la longitud promedio en bits de cada mensaje de la fuente. En nuestro caso como el archivo tiene 6 caracteres y el promedio es 1.4591 la longitud total del archivo de acuerdo a la entropía es $6 * 1.4591 = 8.7549\text{bits}$ es decir que la **cantidad de información** de nuestro archivo es de casi 9 bits. Esto es muy importante ya que es una forma de aproximar la complejidad del mismo de acuerdo a un cierto modelo probabilístico.

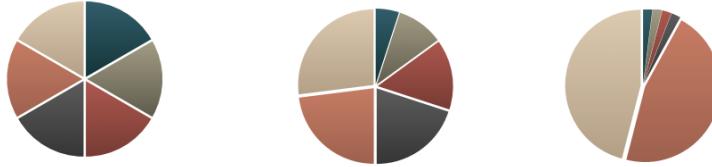


Figure 4.1: Tres entropías diferentes.

La entropía se maximiza cuando todos los mensajes son equiprobables y se minimiza cuando un mensaje tiene probabilidad 1 es decir que se trata de una fuente determinística. Esto nos lleva a varias conclusiones importantes. En primer lugar un archivo aleatorio es aquel que maximiza la entropía, si los mensajes tienen 8 bits en un archivo aleatorio la probabilidad de todos sus mensajes es $1/256$ y la entropía es 8 es decir que en promedio necesitamos 8 bits por cada 8 bits del archivo. Esto implica que los archivos aleatorios no se pueden comprimir, contienen información pura. Puede parecer contradictorio decir que algo completamente aleatorio es información pero para nuestro tema la información es desorden y el orden implica redundancia.

En la figura 4.1 vemos tres diferentes distribuciones de probabilidades. La entropía es máxima para la distribución de la izquierda que es la que tiene probabilidades mas parejas. La entropía mas chica corresponde a la distribución de la derecha en donde hay dos mensajes con muy alta probabilidad y finalmente la distribución en el medio esta en el medio de las otras dos.

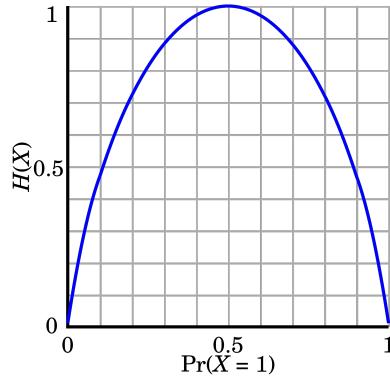


Figure 4.2: Three different entropies

La figura 4.2 nos muestra la entropía para dos mensajes en donde la probabilidad de uno es uno menos la probabilidad del otro, por lo tanto podemos graficar la entropía en función de solo una probabilidad. Como podemos ver la entropía es máxima para $P = 0.5$ es decir cuando ambos mensajes son equiprobables. Cuando $P = 0$ o $P = 1$ la entropía es cero ya que la fuente es determinística.

4.2.5 Entropía Conjunta

Vamos a suponer ahora que tenemos un set de datos con dos atributos "X" que puede tomar valores "R", "G" o "B" e "Y" que puede tomar valores "Y" o "N". La tabla 3.2 nos muestra nuestro set de datos:

X	Y
R	Y
G	N
R	Y
R	N
G	Y
B	Y

Table 4.2: Set de datos para entropía conjunta

A modo de repaso vamos a calcular primero la entropía de cada columna es decir de los atributos "X" e "Y" de forma independiente:

$$H(X) = H[3/6; 2/6; 1/6] = 1.4591$$

$$H(Y) = H[4/6; 2/6] = 0.9182$$

Para calcular la entropía de X simplemente tomamos las probabilidades de los tres valores posibles para X: R=3/6 G=2/6 y B=1/6. Para Y hay dos valores posibles con probabilidad Y=Y 4/6 e Y=N 2/6.

Vamos a definir la entropía conjunta como la entropía de todos los pares (X,Y) que ocurren en nuestro set de datos:

Definition 4.9 (Entropía Conjunta)

$$H(X, Y) = - \sum_{x \in X} \sum_{y \in Y} P(x, y) \log_2 P(x, y)$$

En nuestro ejemplo tenemos los siguientes pares (x,y) y sus probabilidades:

```

RY = 2/6
RN = 1/6
GY = 1/6
GN = 1/6
BY = 1/6

```

Entonces $H(X, Y) = H[2/6; 1/6; 1/6; 1/6; 1/6] = 2.251629$ La entropía conjunta mide la cantidad de información contenida por ambas variables al mismo tiempo. Es decir la entropía del set de datos completo.

4.2.6 Entropía Condicional

La entropía condicional mide la entropía de una variable aleatoria sujeta a que conocemos el valor de otra. En nuestro caso sería equivalente a medir la entropía de "X" sabiendo el valor de "Y" o la entropía de "Y" conociendo el valor de "X".

Una forma intuitiva de entender la entropía condicional es pensar que tenemos dos variables sobre personas: edad y peso. Necesitamos una cierta cantidad de bits para transmitir la edad y otra cantidad para el peso. Pero si conocemos la edad entonces necesitamos menos información para el peso porque una persona de 5 años de edad no puede pesar 90 kilos mientras que una persona de 40 si. Entonces conocer la edad nos restringe la variabilidad de la variable peso y necesitamos menos bits para transmitirla.

Definition 4.10 (Entropía Condicional)

$$H(Y|X) = \sum_{x \in X} P(x) H(Y|X = x)$$

Veamos como calcular $H(Y|X)$ y $H(X|Y)$ para nuestro ejemplo. Para calcular $H(Y|X)$ tenemos que calcular la entropía de Y fijando el valor de X y luego multiplicar dicha entropía por la probabilidad de X, X puede tomar 3 valores posibles R,G, B con probabilidad 3/6,2/6 y 1/6 respectivamente. Entonces:

$$H(Y|X) = \frac{3}{6} H(Y|X = R) + \frac{2}{6} H(Y|X = G) + \frac{1}{6} H(Y|X = B)$$

Para $X=R$ Y toma 2 veces valor "Y" y una vez valor "N" por lo tanto $H(Y|X=R) = H[2/3;1/3]$

$$H(Y|X) = \frac{3}{6}H[2/3;1/3] + \frac{2}{6}H[1/2;1/2] + \frac{1}{6}H[1]$$

$$H(Y|X) = 0.7929$$

Recordemos que $H[1] = 0$ por ser determinístico es decir que cuando $X=B$ la variable Y siempre toma el mismo valor.

De la misma forma podemos calcular $H(X|Y)$

$$H(X|Y) = \frac{4}{6}H[2/4;1/4;1/4] + \frac{2}{6}H[1/2;0;1/2] = 1.333$$

Ahora que hemos calculado $H(X)$, $H(Y)$, $H(X,Y)$, $H(X|Y)$ y $H(Y|X)$ tenemos que analizar la relación que existe entre estas entropías.

Theorem 7 (Regla de la Cadena)

$$H(X,Y) = H(X) + H(Y|X)$$

Antes de realizar una demostración formal es conveniente analizar lo que el teorema nos dice. La entropía conjunta de dos variables es igual a la entropía de una de esas variables mas la entropía condicional de la otra variable sujeta a la primera. Esto tiene mucho sentido ya que si conocemos la entropía de X entonces para calcular la entropía de Y podemos considerar fijo el valor de X y eso nos deja en la entropía condicional.

Demostración:

$$\begin{aligned} H(X,Y) &= - \sum_{x \in X} \sum_{y \in Y} P(x,y) \log_2 P(x,y) \\ H(X,Y) &= - \sum_{x \in X} \sum_{y \in Y} P(x,y) \log_2 P(x)(y|x) \\ H(X,Y) &= - \sum_{x \in X} \sum_{y \in Y} P(x,y) \log_2 P(x) - \sum_{x \in X} \sum_{y \in Y} P(x,y) \log_2 P(y|x) \\ H(X,Y) &= - \sum_{x \in X} P(x) \log_2 P(x) - \sum_{x \in X} \sum_{y \in Y} P(x,y) \log_2 P(y|x) \\ H(X,Y) &= H(X) + H(Y|X) \end{aligned}$$

Por lo tanto:

$$H(X,Y) = H(X) + H(Y|X) = H(Y) + H(X|Y)$$

4.2.7 Información Mutua

Cuando las variables X e Y son independientes es válido decir que $H(X, Y) = H(X) + H(Y)$ ya que $H(X|Y) = H(X)$ y $H(Y|X) = H(Y)$ cuando las variables no son independientes entonces comparten cierta cantidad de información que definimos como información mutua.

Definition 4.11 (Información Mutua)

$$\begin{aligned} I(X, Y) &= H(Y) - H(Y|X) \\ I(X, Y) &= H(X) - H(X|Y) \\ I(X, Y) &= I(Y, X) \end{aligned}$$

Para calcular la información mutua entre dos variables podemos simplemente calcular la entropía de una variable, por ejemplo $H(X)$ y luego restarle la entropía de la otra variable fijando la primera. Lo que queda tiene que ser la información compartida por ambas variables.

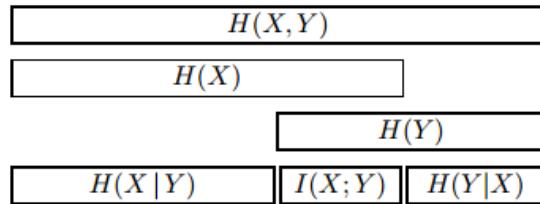


Figure 4.3: Entropias

La figura 4.3 ilustra la relación que existe entre la entropía, entropía conjunta, entropía condicional e información mutua. Tener presente este diagrama nos ayudará a verificar los cálculos que hacemos.

Para nuestro problema por ejemplo tenemos los siguientes valores:

$$\begin{aligned} H(X, Y) &= H(Y, X) = 2.2516 \\ H(X) &= 0.918295 \\ H(Y) &= 1.4591 \\ H(X|Y) &= 0.7929 \\ H(Y|X) &= 1.3333 \\ I(X, Y) &= I(Y, X) = 0.12567 \end{aligned}$$

Podemos verificar que se cumplen todas las condiciones del diagrama:

$$H(X, Y) = H(X) + H(Y|X)$$

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

$$H(X) + H(Y) - I(X, Y) = H(X, Y)$$

etc..

4.2.8 Ejemplo

Veamos un ejemplo del cálculo de las diferentes entropías sobre un archivo de texto.

Sea el siguiente archivo:

ARCHIVO:

aabacbbabbb

BIGRAMAS:

aa

ab

ba

ac

cb

bb

ba

ab

bb

bb

bb

bb

Podemos tomar como "X" al primer carácter de cada bigrama siendo "Y" el segundo carácter de cada bigrama.

	a	b	c	
a	1/10	2/10	1/10	4/10
b	2/10	3/10	0	5/10
c	0	1/10	0	1/10
	3/10	6/10	1/10	

Table 4.3: Probabilidades de los bigramas

Es bueno comprobar que todas las probabilidades sumen 1. El acumulado por filas son las probabilidades para la variable "X" y el acumulado por columnas son las probabilidades para la variable "Y". En la lista de bigramas podemos comprobar que hay 10 bigramas en total y 4 de ellos comienzan con "a" por eso la probabilidad de "a" para la variable X es 4/10, la probabilidad de "b" 5/10 y la probabilidad de "c" 1/10. De la misma forma analizando la segunda columna

de los bigramas que es la variable "Y" tenemos tres "a" (3/10) seis "b" (6/10) y una "c" (1/10).

Con esta tabla estamos en condiciones de calcular todas las entropías que hemos aprendido.

$$H(X) = H[4/10; 5/10; 1/10] = 1.3610$$

$$H(Y) = H[3/10; 6/10; 1/10] = 1.2955$$

$$H(X, Y) = H[1/10; 2/10; 1/10; 2/10; 3/10; 1/10] = 2.4464$$

$$H(X|Y) = 3/10H[1/3; 2/3] + 6/10H[2/6; 3/6; 1/6] + 1/10H[1] = 1.1510$$

$$H(Y|X) = 4/10H[1/4; 1/4; 2/4] + 5/10H[2/5; 3/5] + 1/10H[1] = 1.085$$

$$I(X, Y) = H(X) - H(X|Y) = 0.21$$

4.2.9 Entropía Relativa

La entropía relativa mide la distancia (diferencia) entre la distribución de dos variables aleatorias. Es una medida de la ineficiencia que tenemos si asumimos que una variable tiene una cierta distribución "P" cuando en realidad tiene una distribución "Q"

Definition 4.12 (Divergencia de Kullback Leibler (entropía relativa))

$$D(P||Q) = \sum_{x \in X} P(x) \log_2 \frac{P(x)}{Q(x)}$$

Como la divergencia de Kullback Leibler es la distancia entre una distribución y otra vale decir que:

$$H(X) = H(Y) + D(Y||X)$$

Es decir que si queremos codificar una cierta variable aleatoria "X" pero usando la distribución de otra la divergencia de KL nos da el precio en bits que tenemos que pagar.

Veamos un ejemplo, las tres variables tienen que tener los mismos "x" es decir iguales valores posibles, sino no tiene sentido calcular la divergencia. Sean X,Y,Z variables aleatorias con tres valores posibles: A,B,C con la siguiente distribución.

$$\begin{aligned} X &= [1/5; 2/5; 2/5] \\ Y &= [3/5; 1/5; 1/5] \\ Z &= [6/8; 1/8; 1/8] \end{aligned}$$

Calculemos las diferentes divergencias:

$$D(X||Y) = 1/5 \log_2 \frac{1/5}{3/5} + 2/5 \log_2 \frac{2/5}{1/5} + 2/5 \log_2 \frac{2/5}{1/5} = 0.4830$$

$$D(Y||X) = 3/5 \log_2 \frac{3/5}{1/5} + 1/5 \log_2 \frac{1/5}{2/5} + 1/5 \log_2 \frac{1/5}{2/5} = 0.5510$$

$$D(X||Z) = 1/5 \log_2 \frac{1/5}{6/8} + 2/5 \log_2 \frac{2/5}{1/8} + 2/5 \log_2 \frac{2/5}{1/8} = 0.9611$$

$$D(Z||X) = 6/8 \log_2 \frac{6/8}{1/5} + 1/8 \log_2 \frac{1/8}{2/5} + 1/8 \log_2 \frac{1/8}{2/5} = 1.0106$$

$$D(Y||Z) = 3/5 \log_2 \frac{3/5}{6/8} + 1/5 \log_2 \frac{1/5}{1/8} + 1/5 \log_2 \frac{1/5}{1/8} = 0.0781$$

$$D(Z||Y) = 6/8 \log_2 \frac{6/8}{3/5} + 1/8 \log_2 \frac{1/8}{1/5} + 1/8 \log_2 \frac{1/8}{1/5} = 0.0718$$

Como puede verse la divergencia de KL no es simétrica sin embargo $D(X||Y)$ es parecido a $D(Y||X)$ por lo que puede aceptarse como una métrica válida. En nuestro ejemplo las distribuciones mas similares son "Y" y "Z" y puede verse que esto tiene sentido ya que ambas asignan mayor peso al primer elemento "A" y menor peso a los demás elementos.

Esta divergencia es una medida interesante sobre la diferencia que existe en dos distribuciones de probabilidades diferentes para los mismos datos.

4.2.10 Entropía Cruzada

Al cálculo de la entropía de una variable usando la distribución de otra se lo suele llamar "*entropía cruzada*" (cross entropy), se deriva de la siguiente forma:

$$H_x(P, Q) = - \sum P(x_i) \log_2(Q(x_i)) \quad (4.1)$$

$H_x(P, Q)$ siempre es mayor o igual a $H(P)$ ya que estamos usando una distribución que sabemos no es la óptima. Esta medida puede usarse como métrica de error en varios algoritmos en donde el objetivo es acercarnos lo mas posible a la distribución óptima (P)

4.3 Teoría de la Compresión de Datos

Vamos a empezar ahora a analizar la teoría detrás de la compresión de datos. Lo que queremos es dado un archivo o string encontrar la forma de representarlo usando la menor cantidad de bits posibles. Para hacer esto en general podemos dividir el problema de comprimir datos en dos etapas:

1. Modelar los datos
2. Codificar los datos

4.3.1 Modelar los Datos

Modelar es la parte mas interesante y también la mas difícil de la compresión de datos, el objetivo es encontrar un modelo que describa al archivo o string que queremos representar de la forma mas compacta posible. Por ejemplo un modelo puede ser repetir n veces "AB" si el archivo es "ABABABAB...", otro modelo puede ser emitir los dígitos de π a partir del dígito 1711. Está claro que darse cuenta cuál es el mejor modelo para un archivo es un problema de inteligencia artificial y de hecho es un problema intractable, es decir que no tiene solución, esto puede demostrarse simplemente por reducción a la complejidad de Kolmogorov. Por lo tanto la tarea de un compresor es encontrar el mejor modelo posible. Diferentes algoritmos de compresión de datos modelan al archivo en base a diferentes técnicas, algunos se basan en modelos probabilísticos que modelan la probabilidad de ocurrencia de cada carácter o incluso cada bit del archivo. Otros se basan en encontrar secuencias repetidas y reemplazarlas por un puntero a la ocurrencia anterior, otros algoritmos usan técnicas bastante insólitas pero que funcionan muy efectivamente y finalmente algunos algoritmos se basan en combinar muchos modelos. Como hemos comentado en el capítulo uno estos algoritmos que son ensambles de varios modelos son los que hoy en día logran los mejores niveles de compresión. En este capítulo veremos los principales algoritmos de compresión pero antes vamos a hablar de la codificación.

4.3.2 Codificar los Datos

Codificar implica llevar un modelo a una secuencia de bits, para generalizar podemos decir que un modelo convierte un archivo en una secuencia de símbolos y que la codificación de los símbolos implica encontrar un código binario para cada uno de ellos. Como sabemos la codificación óptima para un símbolo de probabilidad P_i es $-\log_2 P_i$, lo que no sabemos, hasta ahora es como generar los códigos. Vamos a ver dos formas de codificar: en primer lugar códigos de Huffman y finalmente compresión aritmética.

4.3.3 No Existe un Compresor que Pueda Comprimir Cualquier Archivo

Es sencillo demostrar que no existe un compresor que pueda comprimir cualquier tipo de datos, esto se logra mediante el llamado "counting argument" o "pigeon-hole principle". Sin perder generalidad pensemos en archivos de 3 bits, existen en total 8 de estos archivos: 000,001,010,011,100,101,110 y 111. Si un compresor comprime cualquier archivo entonces debería comprimir cada uno de estos archivos en 1 bit, pero solo existen 4 archivos de 2 bits 00,01,10 y 11. Por lo tanto dos o más de los archivos de 3 bits se comprimen en el mismo archivo de 2 bits lo cual implica que será imposible descomprimirlos. Este argumento se extiende fácilmente a " n " bits es decir a cualquier archivo. Existen menos archivos

comprimidos que archivos posibles por lo tanto no puede haber una relación 1 a 1 entre ambos lo cual demuestra que tal algoritmo no podría descomprimirse.

Otra forma simple de entender el counting argument es pensar que un algoritmo capaz de comprimir cualquier archivo, aunque sea 1 bit puede aplicarse recursivamente a la salida del compresor de forma tal que cualquier archivo podrá quedar comprimido finalmente en un bit lo cual es totalmente absurdo.

4.4 Códigos de Huffman

Los códigos de Huffman se usan para representar un set de caracteres en base a sus probabilidades usando códigos de una cierta longitud entera de bits. Se puede demostrar que para códigos de longitudes enteras los códigos de Huffman son óptimos, es decir que aproximan de la mejor forma posible a la longitud ideal de cada código que como sabemos es $-\log_2 P_i$.

4.4.1 Huffman Estático

En el método estático vamos a hacer dos pasadas a los datos para generar los códigos, una primer pasada para contar la cantidad de veces que aparece cada carácter y una segunda pasada para generar los códigos.

La construcción de los códigos de Huffman puede hacerse tanto a partir de la probabilidad de cada mensaje como también a partir de sus frecuencias ya que en definitiva $P_i = f_i / \sum_i f_i$. Vamos a mostrar el algoritmo a partir de un ejemplo:

Sea el siguiente archivo:

baceabbacd

FRECUENCIAS:

a =	3
b =	3
c =	2
d =	1
e =	1

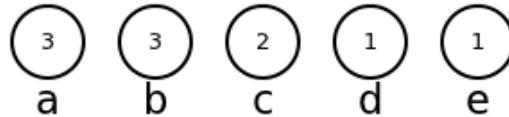


Figure 4.4: Huffman estático estado inicial

En primer lugar convertimos cada mensaje y su frecuencia en un nodo, tenemos tantos nodos como mensajes distintos, en este caso cinco mensajes y cinco

nodos. En cada paso vamos a unir los dos códigos de menor frecuencia formando un nuevo nodo cuya frecuencia será igual a la suma de los nodos que unimos.

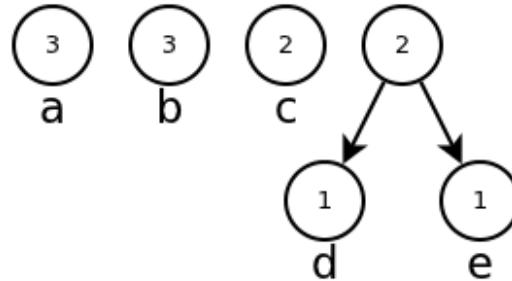


Figure 4.5: Huffman estático.

Notemos que luego del primer paso (fig 3.5) nos quedan cuatro nodos con frecuencias 3,3,2 y 2 respectivamente. El algoritmo continua hasta que queda formado un solo nodo.

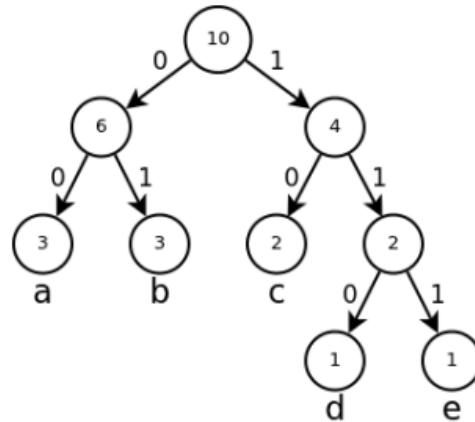


Figure 4.6: Huffman estático.

El resultado final es un árbol binario. Rotulando la rama izquierda como "0" y la rama derecha como "1" por convención el camino desde la raíz hasta cada hoja nos genera el código binario de cada mensaje.

```

a = 00
b = 01
c = 10
  
```

```
d = 110
e = 111
```

Los códigos de Huffman son siempre prefijos y por lo tanto decodificables e instantáneos. El algoritmo es muy simple, el único detalle es que ocurre en caso de empates, es decir si los nodos con menor frecuencia son tres o mas. La respuesta es que podemos tomar cualquier criterio para elegir dos de estos tres nodos siempre y cuando no sea un criterio al azar o dependiente del entorno del programa. La clave es que el descompresor tiene que ser capaz de generar el mismo árbol a partir de la misma lista de frecuencias.

Si quisieramos usar Huffman estático como un algoritmo de compresión deberíamos emitir en el archivo comprimido la tabla de frecuencias y a continuación el resultado de reemplazar cada carácter del archivo por su código correspondiente.

ARCHIVO:
baceabbacd

COMPRIMIDO:
FREQS + 0100101110001010010110

Si el archivo no tuviera una cantidad de bits múltiplo de 8 entonces habría que completar (padding) con los bits que hagan falta, en este caso podemos hacer el padding con cualquier cosa ya que no hay posibilidad de que el descompresor se confunda e intente descomprimir el padding. Esto es porque la tabla de frecuencias le da al descompresor la cantidad total de bytes en el archivo original, entonces una vez que ha descomprimido esa cantidad de caracteres simplemente se detiene y descarta los bits que sobran.

Para descomprimir el proceso es muy simple. El descompresor lee primero la tabla de frecuencias. Genera el mismo árbol que el comprresor y luego va procesando bit a bit el archivo comprimido. Comenzando por la raíz por cada bit que lee se mueve a derecha o izquierda en el árbol según el bit leído. Cuando llega a una hoja emite ese carácter al archivo de salida y vuelve a la raíz.

En general Huffman estático no tiene muy buen nivel de compresión por lo que no se lo usa como algoritmo de compresión stand-alone. Esto no quiere decir que los códigos de Huffman no sean importantes ya que se los usa en muchísimos algoritmos de compresión como un método eficiente para generar códigos a partir de las probabilidades de un conjunto de símbolos.

4.4.2 Huffman Dinámico

La versión dinámica del algoritmo de Huffman tiene como objetivo realizar una única pasada al archivo o string a comprimir. Todos los métodos que llamamos dinámicos comparten esta característica. La necesidad de un algoritmo dinámico puede darse por diversos factores. En telecomunicaciones, por ejemplo, no es posible realizar una pasada a los datos para luego comprimirlos y transmitirlos sino que es necesario ir comprimiendo a medida que los datos lleguen. Mas

adelante veremos otros motivos que desencadenan la necesidad de un algoritmo dinámico.

Comprimir en forma dinámica usando Huffman es sencillo. En principio debemos comenzar suponiendo que todos los mensajes **posibles** de la fuente son equiprobables y en base a ello construir un árbol. Para un archivo esto implica un árbol de 256 elementos ya que cualquier combinación de 8 bits es probable. Para suponer todos los mensajes equiprobables simplemente le asignamos a todos los mensajes frecuencia igual a 1, notemos que no sería lo mismo usar frecuencia igual a 0 porque entonces el árbol de Huffman podría quedar desbalanceado ya que $0+0=0$ en todos los pasos y un árbol desbalanceado contradice la hipótesis que todos los mensajes son equiprobables.

Una vez construido el árbol inicial procedemos a comprimir el primer carácter del archivo. Luego se incrementa la frecuencia de dicho carácter en el árbol y con la nueva lista de frecuencias se construye un árbol nuevo. Este proceso se repite hasta que no quedan caracteres por comprimir en el archivo original.

El árbol final del método de Huffman dinámico es igual al árbol inicial del método estático, es decir que en el método dinámico empezamos con un árbol balanceado y vamos modificándolo hasta llegar al óptimo. La conclusión es que el método dinámico nunca puede comprimir mejor que el método estático, en el mejor de los casos y en forma excepcional es igual, por ejemplo si el archivo es completamente aleatorio entonces el árbol inicial, balanceado, ya es el óptimo. En todos los demás casos el método dinámico comprime un poco menos que el método estático pero hace una sola pasada al archivo a comprimir y no necesita transmitir la tabla de frecuencias.

El descompresor funciona exactamente igual que el compresor. Comienza suponiendo todos los caracteres equiprobables y construyendo un árbol para ellos, luego descomprime bit a bit hasta llegar a una hoja del árbol. Emite el carácter encontrado en dicha hoja, aumenta la frecuencia de ese carácter y reconstruye el árbol en base a las nuevas frecuencias.

Es de destacar que el descompresor desconoce la longitud original del archivo por lo que si el compresor tuvo que usar padding para completar una longitud entera de bits el descompresor podría intentar descomprimir el padding generando caracteres en el archivo descomprimido que no existen en el archivo original. Una buena forma de evitar esto es haciendo que el compresor use un símbolo especial "EOF" que emite solo cuando el archivo se ha terminado, este símbolo es simplemente un mensaje mas es decir que los árboles tendrán 257 nodos en lugar de 256. Cuando el descompresor "descomprime" el "EOF" sabe que el archivo se ha terminado y descarta los bits restantes si existieran.

4.4.3 Modelos de Orden Superior

Hasta el momento hemos calculado la probabilidad de ocurrencia de cada carácter en base a su frecuencia, así si el carácter "A" aparece 4 veces en un archivo de 20 caracteres su probabilidad será $4/20$. Este es un modelo basado en la suposición de que la ocurrencia de un determinado carácter en un archivo es independiente de los caracteres anteriores. Como podemos suponer esta su-

posición no es válida. Si el carácter anterior es una "q" la probabilidad de una "u" será muy alta, mucho más alta que la probabilidad de la misma "u" si el carácter anterior es por ejemplo una "o". En un modelo de orden "n" la frecuencia de cada carácter se calcula en base a los "n" caracteres anteriores. Cada combinación de "n" caracteres del archivo nos proporciona un **contexto** y en base a dicho contexto calculamos la probabilidad del carácter siguiente.

Veamos un caso en el cual usamos Huffman estático de orden 1 es decir que calculamos la frecuencia de cada carácter tomando como contexto el carácter anterior.

ABCABCDABC

Tabla de Frecuencias:

Contexto A: B = 3
 Contexto B: C = 3
 Contexto C: A=1 D=1
 Contexto D: A=1

Veamos ahora como comprimiríamos el archivo usando Huffman estático. En primer lugar hay que transmitir la tabla de frecuencias. Luego como estamos usando orden 1 el primer carácter en general se copia sin comprimir porque no tiene contexto. En un modelo de orden 2 se copian los dos primeros caracteres y así sucesivamente. Es decir que nuestro archivo comprimido comienza con la tabla de frecuencias y una "A"

TABLA FREQS + A

Ahora tenemos que comprimir a cada carácter en base a su contexto, es decir que vamos a tener un árbol de Huffman para cada contexto posible. Pero en varios contextos tenemos un único carácter es decir que el contexto es determinístico. En estos casos no hace falta generar bit alguno para representar al carácter porque la tabla de frecuencias ya contiene la información necesaria. Conociendo la tabla de frecuencias y conociendo el carácter "A" sabemos que el siguiente tiene que ser una "B", como el anterior es ahora "B" el siguiente tiene que ser "C", como el anterior es "C" el siguiente puede ser "A" o "D", aquí si necesitamos un árbol de Huffman en este caso un árbol de solo dos nodos por lo que el código de "A" es decir que para comprimir la "A" que viene luego de la "C" emitimos un bit por ejemplo 0. Un detalle muy curioso es que cuando llegamos nuevamente a tener la "C" como contexto y en el archivo viene "D" no hace falta generar nada porque el descompresor sabe que solo son posibles A y D con frecuencia 1 cada una y como "A" ya fue emitido necesariamente ahora tiene que venir una "D". Por lo tanto el archivo final comprimido es el siguiente:

TABLA FREQS + A1

Donde el 1 es un bit, eventualmente según como se codifique la tabla de frecuencias harán falta algunos bits de padding que pueden ser aleatorios ya

que la tabla de frecuencias le informa al descompresor la cantidad de caracteres que tenía el archivo original.

Un compresor dinámico de orden 1 no necesitaría la tabla de frecuencias pero deberá empezar suponiendo que todos los caracteres son equiprobables en cada contexto es decir que al principio para el contexto "A" tendremos a los 256 bytes posibles con frecuencia 1. Al leer la primera "B" actualizamos el árbol pero el nuevo árbol solo lo usaremos la próxima vez que el carácter anterior sea "A". Es decir que cada contexto tendrá su propio árbol que representa la ocurrencia de los caracteres en base al carácter anterior.

El problema de frecuencia cero

Analizaremos ahora que sucede a medida que usamos modelos de orden cada vez mayor, vamos a ver que tanto los compresores estáticos como los dinámicos tienen un problema con esto.

Empecemos por los compresores estáticos. Estos son muy eficientes en cuanto a la cantidad de bits generados por cada carácter pero a medida que aumenta el orden las tablas de frecuencias dejan de ser insignificantes y se vuelven un factor importante. En un modelo de orden 0 hace falta una única tabla de 256 caracteres, esta es despreciable en función del tamaño, un modelo de orden 1 necesita 256 tablas de 256 caracteres cada uno lo cual está en el orden de los 64K, un modelo de orden 2 está en el orden de los 16Mb y para orden 3 ya pasamos a 4Gb (!). Esto quiere decir que para que tenga sentido comprimir un archivo con un compresor estático de orden 3 necesitamos que el archivo sea tan grande que los 4Gb necesarios para almacenar las tablas de frecuencias sean despreciables. La conclusión es que a partir de orden 2 es necesario que nuestro compresor sea dinámico ya que de lo contrario el peso de las tablas de frecuencias domina sobre el nivel de compresión que pueda alcanzar el método.

Sin embargo los métodos dinámicos de orden superior también tienen serios problemas. Las tablas de frecuencias no son necesarias, el compresor va a emitir únicamente el resultado de codificar cada carácter en función de su contexto. Supongamos que usamos un modelo muy sencillo de orden 1 es decir que codificamos cada carácter en función del anterior. Imaginemos que el carácter anterior es una "q" y que estamos a punto de comprimir el carácter "u". La primera vez que se da esta situación la probabilidad de la "u" será 1/256, supongamos también que luego de "q" no ocurre otra cosa que no sea "u", la segunda vez que veamos "u" su probabilidad será 2/257 luego 3/258, 4/259, etc. Como podemos ver lentamente la probabilidad de la "U" va aumentando pero el compresor tarda muchísimo en darse cuenta que con contexto "q" la probabilidad de la "u" deberá ser casi 1. Este problema se llama "problema de frecuencia cero" ya que su origen se debe a que no podemos en el contexto "q" tener caracteres con frecuencia 0 ya que si estos ocurrieran no los podríamos representar. Sin embargo a efectos de que el compresor aprenda esto es precisamente lo que necesitamos ya que inicializando todo con frecuencia 1 el compresor necesitaría de un archivo enorme para tener tiempo de aprender y comprimir eficientemente. Es decir que tanto los métodos estáticos como los dinámicos necesitan de archivos cada vez

mas grandes para funcionar con orden cada vez mayor.

El problema de frecuencia cero lo vamos a resolver por primera vez en el algoritmo PPMC que veremos luego de introducir la compresión aritmética.

4.5 Compresión Aritmética

Un compresor aritmético es capaz de comprimir los mensajes de una fuente en una cantidad no-entera de bits, gracias a esta capacidad los compresores aritméticos son óptimos dada una cierta distribución de probabilidades, es decir que son los que mejor aproximan a la longitud ideal de los códigos que es $-\log_2 P_i$.

Un compresor aritmético transforma un archivo en un número en el intervalo $[0, 1)$, la precisión del número determina la longitud del archivo comprimido. Al igual que el método de Huffman un compresor aritmético puede ser estático, dinámico y usar modelos de orden superior.

Vamos a mostrar el funcionamiento de un compresor aritmético mediante un sencillo ejemplo en el cual usamos un compresor aritmético estático de orden cero para el siguiente archivo:

AAABBBAC

Al ser estático nuestro compresor calcula la probabilidad de cada carácter haciendo una pasada al archivo: $P(A) = 1/2$; $P(B) = 3/8$; $P(C) = 1/8$ con estas probabilidades vamos a dividir el intervalo cero uno en tres segmentos de longitud proporcional a la probabilidad de cada carácter. Es decir que "A" va ir desde 0 (inclusivo) hasta 0.5 (exclusivo), luego tendremos "B" desde 0.5 hasta 0.875 y finalmente "C" desde 0.875 hasta 1. Una vez dividido el intervalo el compresor puede comprimir el primer carácter "A" para ello lo que hace es quedarse con el intervalo que corresponde a la "A" es decir que nuestro nuevo intervalo será $[0, 0.5]$.

En el siguiente paso el compresor tiene que subdividir el intervalo $[0, 0.5]$ con las mismas probabilidades que antes ahora "A" va ir desde 0 a 0.25, "B" va ir desde 0.25 hasta 0.4375 y "C" va ir desde 0.4325 hasta 0.5. No creemos que haga falta explicar los cálculos ya que son muy sencillos, simplemente particionamos el intervalo en tres segmentos de longitud proporcional a la probabilidad de cada carácter. Este proceso se repite hasta que no queden más caracteres en el archivo de entrada.

Una vez que hemos procesado todo el archivo el resultado será cualquier número perteneciente al intervalo final. Antes de explicar algunos detalles técnicos de la compresión veamos el funcionamiento del descompresor.

El descompresor recibe la tabla de frecuencias (por ser estático) y un número, luego puede simplemente revertir el funcionamiento del compresor. Empezando por el intervalo $[0, 1)$ puede subdividirlo de acuerdo a las probabilidades (que sabe por tener la tabla de frecuencias), llegando al mismo estado inicial que el compresor. Luego se fija a qué carácter pertenece el número y emite ese carácter. Por ejemplo si el número fuera 0.0956878662109375 sabemos que el primer

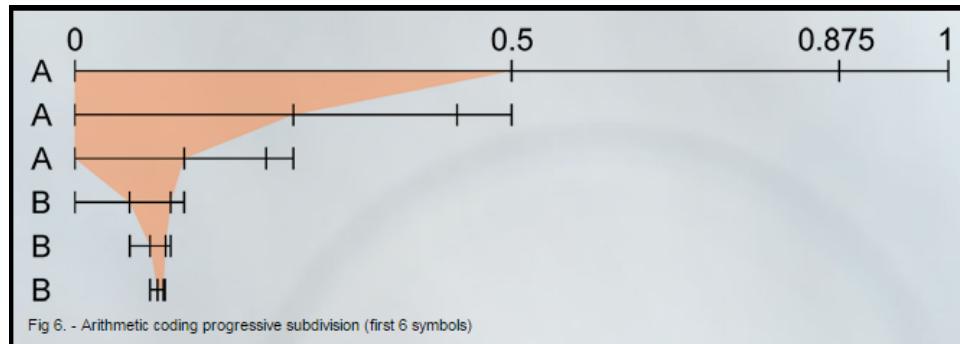


Figure 4.7: Ejemplo de compresión aritmética

caracter tiene que ser "A" ya que el número está dentro del intervalo [0, 0.5) que pertenece a la "A", los demás caracteres no son posibles. El descompresor puede entonces generar una "A" en el archivo de salida. Una vez que ha hecho esto puede quedarse con el intervalo [0, 0.5) y nuevamente particionarlo de acuerdo a las probabilidades de cada carácter. Este proceso se repite hasta que se hayan descomprimido el total de bytes del archivo original, observando que dicha longitud se conoce a partir de la tabla de frecuencias.

La lista de intervalos que el comprimidor produce es la siguiente:

```

Primer Intervalo: [0,1)
Luego de "A" : [0,0.5)
Luego de "A" : [0,0.25)
Luego de "A" : [0,0.125)
Luego de "B" : [0.0625,0.109375)
Luego de "B" : [0.0859375,0.103515625)
Luego de "B" : [0.0947265625,0.096923828125)
Luego de "A" : [0.0947265625, 0.0958251953125)
Luego de "C" : [0.0956878662109375,0.0958251953125)

```

El resultado final es cualquier número dentro del intervalo final por ejemplo 0.0957. Es importante realizar las cuentas y verificar los resultados con la tabla anterior para validar el correcto entendimiento de un comprimidor aritmético, muchos de los algoritmos de compresión que veremos a continuación están basados en codificar símbolos con compresión aritmética.

4.5.1 Implementación y Renormalización

La implementación de un comprimidor aritmético implica solamente llevar dos variables, una para el intervalo superior y otra para el intervalo inferior en binario. Inicialmente el intervalo inferior es 00000000 y el intervalo superior

es 11111111 en la cantidad de bits que se decida usar. Notar que 11111111 representa 0.1111111... que es 1 en binario.

A medida que el compresor avanza va recalculando los intervalos y para no tener problemas de precisión es posible ir emitiendo el número final a medida que se genera, cuando los primeros "n" bits de los dos intervalos son iguales entonces es posible emitir esos bits al archivo de salida y correrlos a izquierda. Por ejemplo si piso=0.101001 y techo=0.101100 entonces es posible emitir los primeros 3 bits 101 al archivo de salida ya que no van a cambiar nunca y quedarnos con los intervalos 001000 y 100111 notar que cuando corremos el intervalo superior completamos con 1s.

Este sencillo procedimiento permite realizar el proceso de compresión aritmética pero hay que tener cuidado con situaciones de underflow. El Underflow puede producirse cuando el piso es de la forma 01... y el techo es de la forma 10... en ese caso puede que la precisión de los números deje de ser la suficiente sin que el primer dígito sea igual para piso y techo. Para evitar esto puede usarse el procedimiento conocido como **renormalización** que consiste en llevar un contador de los dígitos problemáticos y correrlos a izquierda.

Por ejemplo:

SUP:100110
INF:011001

(Renormalizar)

SUP: 110111
INF: 001000 (Contador = 3)

Eventualmente el primer dígito de los intervalos va a coincidir, si el primer dígito converge a 1 entonces además del 1 emitimos tantos 1s como indique el contador y si el primer dígito converge a 0 entonces además del 0 emitimos tantos 0s como indique el contador y lo reseteamos a cero.

4.6 PPMC

PPMC (prediction by partial matching) es un algoritmo que alcanza un excelente nivel de compresión, está basado en modelos de compresión aritmética pero resolviendo el problema de frecuencia cero. La forma en la cual PPMC resuelve el problema de frecuencia cero es muy ingeniosa, en lugar de usar un solo modelo de compresión de orden "n" usa varios modelos desde orden "n" hasta orden -1. Cuando un carácter no aparece en el modelo en el cual lo buscamos el algoritmo emite un símbolo especial (escape) y pasa al modelo de orden inmediato inferior. En cada modelo se inicializa únicamente el escape y se van agregando símbolos a medida que estos son observados en el archivo a comprimir. El modelo de orden -1 es un modelo especial en el cuál se inicializan todos los caracteres mas el símbolo de EOF con frecuencia 1 es decir probabilidad 1/257, de esta forma

se garantiza que cualquier símbolo del archivo original puede ser encontrado en alguno de los modelos.

PPMC es en esencia un compresor aritmético, en cada paso subdivide el intervalo actual de acuerdo al modelo que toque usar y se queda el sub-intervalo que corresponde al símbolo emitido, el ESC tiene su intervalo como cualquier otro símbolo. Luego de emitir se actualiza el modelo por el cual pasamos como en cualquier algoritmo de compresión dinámico.

4.6.1 Ejemplo PPMC

Veamos un ejemplo de PPMC de orden máximo 2. Para el ejemplo vamos a suponer que solo son posibles los caracteres A,B,C esto lo hacemos simplemente por razones de espacio para no tener que incluir 256 elementos en el modelo de orden -1 que es el único que inicialmente contiene todos los caracteres posibles. Los demás modelos están vacíos lo cual equivale a decir que en cada uno de ellos solo existe el escape (ESC) con probabilidad igual a 1.

La tabla 3.4 muestra el estado inicial de PPMC en base a lo planteado.

M-1	M0	M1	M2
A=1/4 B=1/4 C=1/4 EOF=1/4	ESC=1		

Table 4.4: PPMC estado inicial

Vamos a comprimir usando PPMC el siguiente archivo:

ARCHIVO: ABCCCC

Comprimir A

El primer carácter a comprimir es "A", como no tenemos contexto alguno comenzamos en el modelo de orden 0. Nuestro intervalo inicial es el [0,1). En el modelo de orden 0 lo único que existe es un escape por lo tanto el intervalo [0,1) queda ocupado totalmente por el escape, al emitir el escape entonces nuestro intervalo no se modifica. Esto es intuitivo ya que al ser el único símbolo existente en el modelo la probabilidad del escape es 1 y un símbolo de probabilidad 1 tiene longitud ideal $-\log_2(1) = 0$ bits por lo tanto es lógico que no modifique nuestro intervalo, no necesitamos bits para representar que hemos visto un escape.

Pasamos al modelo de orden -1 y en este modelo encontramos A,B,C y EOF con probabilidades 1/4 cada uno, nuestro intervalo queda dividido de la siguiente forma:

Notemos que es arbitrario el orden en el cual aparecen los caracteres. El carácter a emitir es "A" por lo tanto nuestro nuevo intervalo es [0,0.25). Es fundamental notar que cada vez que PPMC "emite" un símbolo lo que realmente

A	[0-0.25)
B	[0.25-0.5)
C	[0.5-0.75)
EOF	[0.75-1)

está haciendo es seleccionando un nuevo intervalo en base a la distribución de probabilidades del modelo en el cual se encuentra y el símbolo a emitir.

Ahora estamos en condiciones de actualizar los modelos por los cuales hemos pasado es decir el modelo 0, el modelo -1 nunca se actualiza.

M-1	M0	M1	M2
A=1/4			
B=1/4	A=1/2		
C=1/4	ESC=1/2		
EOF=1/4			

Table 4.5: PPMC luego de comprimir "A"

Notemos que ahora en el modelo de orden 0 existe la "A" con probabilidad 1/2. En PPMC la frecuencia del escape siempre es igual a la cantidad de caracteres que existen en el modelo, al haber un carácter (A) la frecuencia del escape es 1. Como A tiene frecuencia 1 y ESC tiene frecuencia 1 la probabilidad de ambos es 1/2.

Comprimir (A)B

Nuestro próximo carácter en el archivo es "B" y nuestro contexto es "A", como tenemos sólo un carácter de contexto usamos el modelo de orden 1. En este modelo no existe el contexto "A" por lo tanto emitimos un escape de probabilidad 1 que no le hace nada a nuestro intervalo y pasamos al modelo de orden 0.

En el modelo de orden 0 tenemos "A" y ESC por lo tanto tenemos que emitir un ESC. Para hacerlo tomamos nuestro intervalo actual [0,0.25) y lo dividimos de acuerdo a las probabilidades del modelo:

A	[0-0.125)
ESC	[0.125-0.25)

Como tenemos que emitir un ESC nos quedamos el intervalo [0.125-0.25) que es nuestro intervalo actual. Pasamos ahora al modelo de orden -1. Aquí tenemos A,B,C y EOF con probabilidades 1/4 cada uno. Pero cuidado! Al haber pasado por el modelo de orden 0 sabemos que nuestro carácter a comprimir nunca puede ser "A" ya que de haber sido "A" lo hubiésemos emitido en el modelo anterior en lugar de un escape. Por lo tanto podemos simplemente eliminar la "A" del

modelo de orden -1. Esto se llama **mecanismo de exclusión**. Al eliminar "A" el modelo nos queda $B=1/3$, $C=1/3$, $EOF=1/3$. Procedemos entonces a subdividir nuestro intervalo de acuerdo a estas probabilidades.

B	[0.125-0.1666)
C	[0.1666-0.208333)
ESC	[0.208333-0.25)

De este intervalo tomamos "B" es decir que nuestro nuevo intervalo es [0.125-0.1666). Ahora podemos actualizar los modelos por los que hemos pasado es decir el modelo de orden 1 con contexto "A" y el modelo de orden 0.

M-1	M0	M1	M2
A=1/4	A=1/4	[CTX A]	
B=1/4	B=1/4	B=1/2	
C=1/4	ESC=2/4	ESC=1/2	
EOF=1/4			

Table 4.6: PPMC luego de comprimir "AB"

Notemos que en el modelo de orden 0 la frecuencia del escape es ahora 2 porque tenemos dos caracteres en dicho modelo (A y B). Siempre que se agrega un nuevo carácter al modelo hay que incrementar la probabilidad del ESC.

Comprimir (AB)C

Ahora tenemos que comprimir la "C" y nuestro contexto es "AB" empezamos por lo tanto en el modelo de orden 2 en donde el contexto "AB" está vacío, por lo tanto emitimos un ESC de probabilidad 1 que no cambia nuestro intervalo [0.125-0.1666). Pasamos al modelo de orden 1 en donde el contexto "B" tampoco existe por lo que emitimos otro escape de probabilidad 1 y pasamos al modelo de orden 0. En el modelo de orden 0 tenemos A,B y ESC por lo que tenemos que subdividir nuestro intervalo.

A	[0.125-0.13541666)
B	[0.13541666-0.1458333)
ESC	[0.1458333-0.1666)

Nuestro nuevo intervalo es entonces [0.1458333-0.1666). Estamos ahora en el modelo de orden 1 en donde tenemos que excluir la "A" y la "B" ya que las hemos observado en modelos anteriores en los que emitimos un ESC, por lo tanto ninguno de estos caracteres es posible. Nos quedan solo C y EOF con probabilidad 1/2 cada uno. Aplicando el modelo de orden -1 a nuestro intervalo tenemos:

C	[0.1458333-0.15625)
EOF	[0.15625-0.1666)

De aqui emitimos "C" por lo que nuestro intervalo nuevo es [0.1458333-0.15625).

Tenemos que actualizar todos los modelos por los que pasamos es decir el modelo de orden 2 con contexto "AB", el modelo de orden 1 con contexto "B" y el modelo de orden 0. La tabla de modelos queda de la siguiente forma:

M-1	M0	M1	M2
A=1/4	A=1/6	[CTX A] B=1/2 ESC=1/2	[CTX AB]
B=1/4	B=1/6	[CTX B]	C=1/2
C=1/4	C=1/6	C=1/2	ESC=1/2
EOF=1/4	ESC=3/6	ESC=1/2	

Table 4.7: PPMC luego de comprimir "ABC"

Notemos como PPMC va consumiendo memoria únicamente a medida que lo necesita los modelos solo ocupan el espacio necesario de acuerdo a los caracteres observados, en otras palabras si nunca ocurre "x" luego de "q" entonces nunca tendremos una "x" en el contexto "q" para el modelo de orden 1. Esto es importante ya que de lo contrario los modelos ocuparían una cantidad de memoria inmanejable. De todas formas un PPMC de orden 5 o 6 puede ocupar algunos Gb de RAM lo cual hoy en día afortunadamente es algo manejable.

Comprimir A(BC)C

Hemos comprimido "ABC" y el carácter próximo es otra "C". Nuestro intervalo actual es [0.1458333-0.15625). Como hemos fijado el orden máximo en 2 nuestro contexto es "BC". Por lo tanto comenzamos en el modelo de orden 2 en donde buscamos el contexto BC, el cual no existe por lo que emitimos un ESC de P=1 que no modifica nuestro intervalo y pasamos al modelo de orden 1. Aquí el contexto "C" tampoco existe por lo que emitimos otro escape de P=1 y pasamos al modelo de orden 0 en donde "C" existe por lo que tenemos que subdividir nuestro intervalo de acuerdo a las probabilidades de dicho modelo que son (A=1/6, B=1/6, C=1/6, ESC=3/6).

A	[0.1458333-0.147569444)
B	[0.147569444-0.14930555)
C	[0.14930555-0.151041666)
ESC	[0.151041666,0.15625)

Como emitimos una "C" nuestro nuevo intervalo es [0.14930555-0.151041666]. Actualizamos ahora los modelos por los que pasamos, es decir el contexto "BC" en el modelo de orden 2, el contexto "C" en el modelo de orden 1 y el modelo de orden 0.

M-1	M0	M1	M2
A=1/4 B=1/4 C=1/4 EOF=1/4	A=1/7 B=1/7 C=2/7 ESC=3/7	[CTX A] B=1/2 ESC=1/2 [CTX B] C=1/2 ESC=1/2 [CTX C] C=1/2 ESC=1/2	[CTX AB] C=1/2 ESC=1/2 [CTX BC] C=1/2 ESC = 1/2

Table 4.8: PPMC luego de comprimir "ABCC"

Comprimir AB(CC)C

Ahora tenemos que comprimir "C" con contexto "CC" este contexto no existe en el modelo de orden 2 por lo que se emite un ESC de probabilidad igual a 1, pasamos al modelo de orden 1 y con contexto "C" encontramos la "C". Nuestro intervalo [0.14930555-0.151041666) queda entonces dividido de la forma.

C	[0.14930555-0.1501736111)
ESC	[0.1501736111,0.151041666)

Y como estamos comprimiendo "C" elegimos el intervalo que corresponde a la misma que se convierte en nuestro intervalo actual: [0.14930555,0.1501736111). Actualizamos el contexto "CC" en el modelo de orden 2 y el contexto "C" en el modelo de orden 1. El modelo de orden 0 no se actualiza pues no lo hemos usado.

Comprimir ABC(CC)C

Una vez mas el contexto es "CC" y tenemos que comprimir "C" esta vez encontramos "C" en el modelo de orden 2 con contexto "CC" por lo que subdividimos nuestro intervalo de acuerdo a las probabilidades en dicho modelo:

Nuestro nuevo intervalo es [0.14930555-0.14973958333). Y el único modelo a actualizar es el contexto "CC" en el modelo de orden 2.

M-1	M0	M1	M2
A=1/4 B=1/4 C=1/4 EOF=1/4	A=1/7 B=1/7 C=2/7 ESC=3/7	[CTX A] B=1/2 ESC=1/2 [CTX B] C=1/2 ESC=1/2 [CTX C] C=2/3 ESC=1/3	[CTX AB] C=1/2 ESC=1/2 [CTX BC] C=1/2 ESC = 1/2 [CTX CC] C=1/2 ESC = 1/2

Table 4.9: PPMC luego de comprimir "ABCCC"

C	[0.14930555-0.14973958333)
ESC	[0.14973958333,0.151041666)

M-1	M0	M1	M2
A=1/4 B=1/4 C=1/4 EOF=1/4	A=1/7 B=1/7 C=2/7 ESC=3/7	[CTX A] B=1/2 ESC=1/2 [CTX B] C=1/2 ESC=1/2 [CTX C] C=2/3 ESC=1/3	[CTX AB] C=1/2 ESC=1/2 [CTX BC] C=1/2 ESC = 1/2 [CTX CC] C=2/3 ESC = 2/3

Table 4.10: PPMC luego de comprimir "foooABCCCC"

Comprimir ABCC(CC)

Finalmente llegamos al fin del archivo, es decir que tenemos que comprimir el símbolo EOF, esto es fundamental para decirle al descomprimidor que ya no tiene que seguir procesando el archivo comprimido. Empezamos por el contexto "CC" en el modelo de orden 2 en donde tenemos que emitir un ESC que tiene probabilidad 1/3.

C	[0.14930555-0.1495949074)
ESC	[0.1495949074,0.14973958333)

Nuestro nuevo intervalo es el correspondiente al ESC en dicho modelo es decir: [0.0.1495949074,0.14973958333). Pasamos el modelo de orden 1 en donde tenemos que excluir la "C", al excluir "C" lo único que queda en este mod-

150 CHAPTER 4. COMPLEJIDAD, COMPRESIÓN Y TEORÍA DE LA INFORMACIÓN

elo es el ESC y por lo tanto su probabilidad es 1 es decir que el intervalo no se modifica. Pasamos entonces al modelo de orden 0 en donde tenemos que excluir la "C" y nos queda entonces A=1,B=1,ESC=3. Notar que la frecuencia del ESC no se modifica por aplicar exclusión. Dividimos nuestro intervalo [0.1495949074,0.14973958333) de acuerdo a este modelo.

A	[0.1495949074-0.14962384258)
B	[0.14962384258-0.149652777)
ESC	[0.149652777,0.14973958333)

Seleccionamos el sub-intervalo correspondiente al escape y nuestro nuevo intervalo es: [0.149652777,0.14973958333).

Ahora pasamos al modelo de orden -1 excluyendo "A","B" y "C" que ya fueron observados en modelos anteriores. Como lo único que queda es un EOF su probabilidad es 1 y no hace falta subdividir el intervalo. El EOF siempre va a comprimirse con P=1 en el modelo de orden -1 ya que el modelo de orden 0 va a tener todos los caracteres observados en el archivo hasta su fin.

En definitiva nuestro archivo comprimido va a ser algún número en el intervalo [0.149652777,0.14973958333). Si trabajamos en binario cualquier número dentro de este intervalo sirve, pero como hemos hecho las cuentas en decimal vamos a pasar a binario para mostrar el archivo final comprimido.

Para hacer esto solo necesitamos la tabla de potencias de 2 a partir de -1.

		bit	acum
2^{-1}	0.5	0	0
2^{-2}	0.25	0	0
2^{-3}	0.125	1	0.125
2^{-4}	0.0625	0	0.125
2^{-5}	0.03125	0	0.125
2^{-6}	0.015625	1	0.140625
2^{-7}	0.0078125	1	0.1484375
2^{-8}	0.00390625	0	0.1484375
2^{-9}	0.001953125	0	0.1484375
2^{-10}	0.0009765625	1	0.1494140625
2^{-11}	0.00048828125	0	0.1494140625
2^{-12}	0.00024414062	1	0.14965820312

Table 4.11: Codificación en binario del archivo comprimido

Nuestro primer bit es 0 porque nuestro intervalo es menor a 0.5, ídem para el segundo bit. El tercero bit en cambio tiene que ser 1, el cuarto bit un cero porque sino nos pasamos, etc... Por lo tanto el archivo en binario completo es:

001001100101

Analicemos ahora si este archivo de 12 bits corresponde con lo que esperaríamos de acuerdo a la entropía, esto lo podemos hacer simplemente considerando las probabilidades de cada símbolo que hemos emitido y usando $-\log_2(P_i)$ para estimar su longitud ideal.

archivo	modelo	prob	long	intervalo
A	M0	1	0	[0,1)
A	M-1	1/4	2	[0,0.25)
AB	M1	1	0	[0,0.25)
AB	M0	1/2	1	[0.125,0.25)
AB	M-1	1/3	1.58496250072	[0.126,0.1666)
ABC	M2	1	0	[0.126,0.1666)
ABC	M1	1	0	[0.126,0.1666)
ABC	M0	1/2	1	[0.145833,0.1666)
ABC	M-1	1/2	1	[0.145833,0.15625)
ABCC	M2	1	0	[0.145833,0.15625)
ABCC	M1	1	0	[0.145833,0.15625)
ABCC	M0	1/6	2.58496250072	[0.14930555,0.151041666)
ABCCC	M2	1	0	[0.14930555,0.151041666)
ABCCC	M1	1/2	1	[0.14930555,0.1501736111)
ABCCCC	M2	1/2	1	[0.14930555,0.14973958333)
ABCCCCEO	M2	1/3	1.58496250072	[0.1495949074,0.14973958333)
ABCCCCEO	M1	1	0	[0.1495949074,0.14973958333)
ABCCCCEO	M0	3/5	0.73696559416	[0.149652777,0.14973958333)
ABCCCCEO	M-1	1	0	[0.149652777,0.14973958333)
			13.491851	

Table 4.12: Probabilidades emitidas y sus longitudes

Notemos que el total de bits que hemos emitido (12) es cercano pero inferior a los 13.49 bits que pronosticamos de acuerdo a la entropía, esto se debe a que el 13.49 es el **promedio** en bits para cualquier archivo que tenga la misma distribución de probabilidades que el nuestro, algunos ocuparán 12 , 13 y otros 14 bits. Para archivos de suficiente longitud en general la estimación es muy precisa.

4.6.2 PPMC Aspectos claves

Repasemos algunos aspectos claves de PPMC.

1. Es necesario fijar un orden máximo
2. El modelo de orden -1 tiene todos los caracteres posibles mas el EOF y nunca se actualiza
3. La frecuencia del ESC es igual a la cantidad de caracteres en el modelo

4. Solo se actualizan los modelos por los cuales pasamos y esto se hace DESPUES de haber comprimido cada carácter

4.6.3 Descompresión

El proceso de descompresión en PPMC es sencillo ya que el descompresor sigue los mismos pasos que el compresor. Comienza con los mismos modelos que comienza el compresor y se fija dentro de qué intervalo cae el número que representa al archivo comprimido. De esta forma puede ir emitiendo los diferentes caracteres y de la misma forma que lo hizo el compresor descartando los bits que ya ha descomprimido a medida que va re-escalando los intervalos, siempre siguiendo los pasos del descompresor. Al llegar al símbolo de EOF el descompresor se detiene y el resultado es el archivo original descomprimido.

4.6.4 Orden Máximo para PPMC

Es interesante analizar cuál es el orden óptimo para PPMC. Podemos pensar que a mayor orden mayor nivel de compresión ya que usamos contextos cada vez más largos lo cual nos da mayor precisión al predecir las probabilidades de los caracteres. Sin embargo usar un orden máximo muy alto también implica generar cada vez mayor cantidad de escapes cuando no encontramos el carácter a comprimir en el modelo actual. Eventualmente el peso de los escapes supera a lo que se gana en nivel de compresión y el algoritmo empieza a comprimir peor.

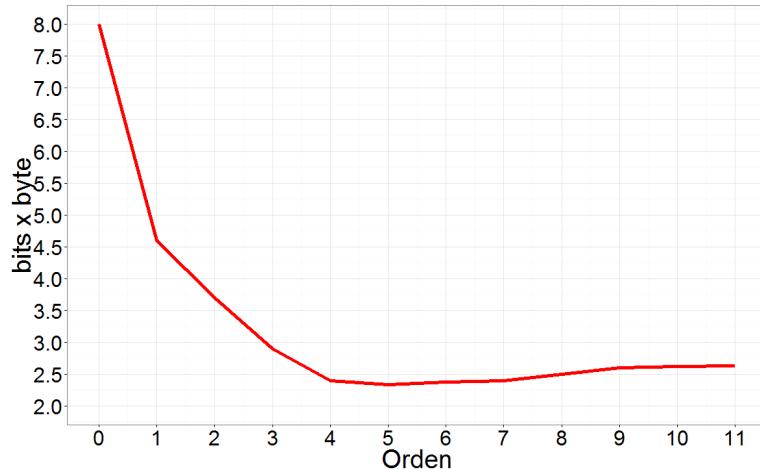


Figure 4.8: Nivel de compresión de PPMC según el orden máximo

El gráfico muestra el nivel de compresión de PPMC para diferentes valores de orden máximo. Como podemos ver el óptimo se encuentra alrededor de 5 o 6 caracteres de contexto. Esto nos da contextos lo suficientemente largos como

para predecir muy bien sin que el peso de los escapes sea significativo. Es por esto que en general suelen usarse contextos de orden 5 o 6 al comprimir con PPMC. Para archivos especiales puede tener sentido usar contextos mas largos o mas chicos según el caso.

4.6.5 Variantes de PPMC

PPMC es la variante "C" de PPM. Casi todas las variantes de PPM giran alrededor de un mismo tema: la frecuencia del escape en cada modelo. Como hemos mencionado el escape debería estar relacionado con la probabilidad de encontrar un símbolo nuevo en el modelo, es decir que ocurra un carácter que aun no hemos visto para un cierto contexto. No hay una forma "exacta" de modelar esta probabilidad por lo que es necesario recurrir a alguna heurística. La heurística conocida como "método C" que es la que usamos en el apunte es una de las que mejor funciona y además es muy simple, ya que solamente consiste en aumentar la frecuencia del escape cada vez que se agrega un nuevo carácter al modelo. Otros variantes de PPM como PPMA, PPMB, PPMD, PPMQ, PPMXC y PPMZ usan diferentes heurísticas para las frecuencias de los escapes [Witten-Bell]. De todos estos métodos el mas interesante es PPMQ porque introduce un concepto muy importante e interesante en compresión de datos conocido como "secondary symbol estimation" (SSE).

4.6.6 PPMQ y SSE

En PPMC la probabilidad del escape es igual a su frecuencia dividida sobre el total de frecuencias. Sin embargo podemos pensar que $1/2$ podría no ser lo mismo que $3/6$ o $10/20$. Cuando la frecuencia del escape es 1 solo existe en el modelo un carácter además del escape, es decir que el modelo es, en cierta forma, determinístico, en este tipo de modelos la probabilidad real del escape debería ser algo menor a $1/2$ mientras que en un modelo en el cual hemos visto 8 caracteres diferentes si la suma total es 16 la probabilidad del escape podría ser algo mayor a $8/16$. En PPMQ se plantea una forma de ajustar la probabilidad del escape de forma tal de reflejar esta intuición.

Para hacer esto introducimos el concepto de "secondary symbol estimation" que como su nombre lo indica consiste en pedir una "segunda opinión" sobre la probabilidad de un cierto símbolo, en este caso de los escapes. La forma en la que PPMQ hace esto es muy simple y elegante. Cada vez que se va a emitir un escape contamos con dos datos sobre el mismo: su frecuencia y la suma total de frecuencias en el modelo. Estos dos datos se indexan mediante una tabla de hash. En esta tabla vamos a contabilizar la cantidad de veces que para esa frecuencia y total se emitió realmente un escape y la cantidad de veces que se emitió otro carácter.

La tabla 3.13 muestra un ejemplo de la estructura que necesita PPMQ, las dos primeras columnas serían la clave de la tabla de hash. Supongamos ahora que estamos en un cierto modelo y tenemos que emitir un escape que tiene frecuencia 2 y el total de frecuencias del modelo es 4. Accedemos a la tabla y

ESC freq	Freq Total	ESC	Other
1	2	1	3
1	3	1	1
2	4	2	3

Table 4.13: PPMQ

vemos que en estas condiciones se ha emitido 2 veces un escape y 3 veces otra cosa. Por lo tanto la probabilidad del Escape es $2/5$ y esta será la probabilidad que usaremos para comprimir. Notemos que en esta tabla la entrada que corresponde a frecuencia= 2 y total =4 reúne estadísticas de TODOS los modelos en los cuales se haya observado esta combinación. Notemos también que cada vez que emitimos un carácter en un modelo (diferente de ESC) tenemos que actualizar la cantidad de "other" en la tabla correspondiente, de esta forma aprendemos que para la frecuencia y total que tenía el escape no hemos emitido un escape sino otro símbolo.

4.6.7 Modelos Generativos Basados en PPMC

Introducimos aquí el fascinante mundo de los modelos generativos. La idea es "aprender" un modelo para un determinado archivo o conjunto de archivos y luego a partir de dicho modelo generar contenido al azar. Por ejemplo podemos comprimir todas las obras de Shakespeare usando PPMC de orden 6. Una vez que hicimos esto tenemos un modelo que representa las obras de Shakespeare y podemos usarlo para generar contenido de forma aleatoria. La forma de hacer esto es muy simple: comenzamos con el modelo de orden 0 y en base a las probabilidades del mismo generamos un carácter al azar. Luego con dicho carácter como contexto vamos al modelo de orden 1 y generamos otro carácter al azar en base a las probabilidades de dicho modelo. Así seguimos generando caracteres de forma aleatoria y el resultado es el siguiente:

Empress' sons
Hath slain to-day!

PROSPERO. How? turn the business for that I love thee now we hold thy time,
Which would it pleas'd;
And take against your writing.

PRINCE JOHN. If th' event o' th' moon,
Whom the King's will fly, like a Northern blasts from this enforce these quondam
Quickly to Sir Nicholas be thy
tongue divine, are gone about me;
her'; and whisp'ring, rounding, though much, is not burden two fair son!
Dogs howl,
Form'd by the hand; les doigts? Ma foi, j'oublie les doigts?

ALICE. Oui, vraiment.

SIR TOBY. Fie that you miss. She'll gallop withal?

HORTENSIO. From th' head.

WARWICK. And so repose myself confounds himself, with another name.

ROMAN. Well, whiles you carry you.

KING EDWARD. And the pard,
Nor build there she remain within

WOODVILLE, Lieutenant's scarf? You
calumny will she lives.
Infirmity,
When English-woman. For what of his legs again?

SECOND WITCH. Fillet of my children;
Than as you

Como podemos ver el algoritmo ha aprendido a escribir algo muy parecido a una obra de Shakespeare pero completamente al azar, notar que incluso aprende a introducir los personajes. El algoritmo tiene algunas limitaciones, por ejemplo no cambia las probabilidades según la cantidad de caracteres que tiene una línea de texto, cuanto mas larga la línea mas probable debería ser que aparezca un punto y un salto de línea. Tampoco sabe que si está en una aposición la coma tiene mayor probabilidad hasta ser generada para cerrar la misma. En definitiva las limitaciones del algoritmo están dadas por su corta memoria, no recuerda lo que ha hecho anteriormente sino que tiene en cuenta únicamente los 6 caracteres anteriores y ese es todo su mundo. Con estas limitaciones el algoritmo es capaz de generar un texto bastante legible lo cual habla muy bien la capacidad de PPMC para modelar archivos. Es lógico suponer que cuanto mejor entiende el algoritmo el archivo que está procesando mejor va a poder generar contenido similar al mismo, esto confirma nuestra idea de que comprimir es en definitiva "entender".

4.6.8 PPMC conclusiones

PPMC es un compresor aritmético dinámico de orden superior que maneja varios ordenes diferentes al mismo tiempo para resolver el problema de frecuencia cero. PPMC es uno de los algoritmos de compresión más importantes, alcanza un nivel de compresión muy competitivo. Algoritmos basados en PPMC son los que mejor nivel de compresión alcanzan para algunos tipos de archivos, por ejemplo textos de gran longitud.

4.7 DMC: Dynamic Markov Compression

DMC [Cormack-Horspool] es un algoritmo basado en comprimir los datos bit por bit, a diferencia de PPMC que procesaba byte a byte. DMC usa un único contexto por cada bit a comprimir a diferencia de PAQ que usa una mezcla de contextos.

Por cada bit a comprimir DMC intenta predecir la probabilidad de que el bit sea 1 o 0 comprimiendo el mismo con compresión aritmética.

Inicialmente DMC comienza con una tabla con todos los posibles contextos de 8 a 15 bits en donde considera al 0 y al 1 equiprobables. A efectos de simplificar vamos a hacer el ejemplo usando 2 bits como contexto.

state	contexto	bit 0 count	next _s tate	bit 1 count	next state
0	00	1	0	1	1
1	01	1	2	1	3
2	10	1	0	1	1
3	11	1	2	1	3

Table 4.14:

Observemos que, por ejemplo, cuando el estado es "10" que es el estado 2, si el próximo bit es un 0 pasamos al estado 0 ya que nuestro contexto va a ser 00 mientras que si el próximo bit es un 1 pasamos al estado 1 porque el contexto es 01.

Esta tabla nos define un modelo de Markov en donde cada estado representa un contexto y las transiciones están dadas por la probabilidad del bit 0 o 1. Los pasos para comprimir son entonces los siguientes:

Partiendo de algún estado inicial:

1. Calcular la probabilidad de que el próximo bit sea 1 o 0
2. Leer el próximo bit del archivo
3. Dividir el intervalo de acuerdo a la probabilidad calculada y el bit leído (compresión aritmética)
4. Incrementar la probabilidad del bit leído en el contexto actual (es fundamental hacerlo **después** de dividir el intervalo)

5. En base al bit leído determinar el próximo estado
6. volver a 1

Notemos que al ser éste un modelo de Markov en ningún momento hace falta buscar el estado en la tabla, las transiciones nos dicen siempre a qué estado movernos en función del bit leído.

4.7.1 Clonación

Así como lo hemos descripto DMC es un compresor aritmético dinámico de orden 2 bits. Lo que nos falta es permitir que el algoritmo aprenda nuevos contextos. Si estamos en el estado "A" y el estado próximo pierde algunos bits entonces si la probabilidad del 1 o el 0 en "A" es superior a la del otro bit de acuerdo a un cierto umbral DMC puede decidir crear un nuevo contexto.

Supongamos el siguiente ejemplo: el estado "A" representa el contexto 11111 cuando vemos un 0 nos movemos a B que representa 110. En el contexto "A" hemos observado 4 ceros y un solo 1. En el contexto B hemos observado 3 ceros y 7 unos. La situación actual es entonces: (solo mostramos los estados A y B)

state	contexto	bit 0 count	next _s state	bit 1 count	next state
A	11111	4	B	1	J
B	110	3	E	7	F

Table 4.15:

Clonamos al estado "C" a partir de B representado el contexto 111110. Las transiciones del estado B serán las mismas que las del estado clonado "C" ya que ese era el estado al cuál íbamos antes desde "A" cuando observábamos un cero.

state	contexto	bit 0 count	next _s state	bit 1 count	next state
A	11111	4	C	1	J
B	110	3	E	7	F
C	111110	4	E	9.33	F

Table 4.16:

La cantidad de ceros de "C" tiene que ser igual a la cantidad de ceros de "A" ya que estamos clonando "A" cuando vemos un cero. Y la probabilidad del 0 en "C" tiene que ser igual a la probabilidad del cero en "B" que era el estado al cuál íbamos antes. Por lo tanto la cantidad de ceros en "C" es 4 y la cantidad de 1s es 9.33. De esta forma "B" y "C" realizan la misma predicción que es $P(0)=0.3$.

Notemos que la condición de que el estado al cuál vamos tenga menos bits evita que volvamos a clonar a "A", porque ahora cuando en "A" vemos un cero vamos al estado "C" que *agrega* un bit al contexto en lugar de quitarlo, por lo tanto no hay que clonar.

En el algoritmo original el criterio para clonar es cuando la transición de A a B es al menos U1 y la cantidad de veces que observamos ese bit en B a partir de cualquier otro estado es al menos U2.

Estos umbrales se puede ir variando a medida que comprimimos para evitar que el algoritmo consuma toda la memoria. Cuando el algoritmo consume toda la memoria el modelo entero se descarta y el algoritmo se re-inicializa a su estado inicial.

DMC es un algoritmo muy interesante ya que con memoria infinita es capaz de generar el modelo completo bit a bit del archivo. Un punto a tener en cuenta es que para que el algoritmo aprenda el modelo es necesario procesar el archivo por lo que el tiempo de entrenamiento puede ser alto.

La construcción de un modelo de Markov byte a byte (PPMC) o bit a bit (DMC) a partir del cuál calcular la probabilidad del próximo carácter o bit suele dar buenos resultados en cuanto al nivel de compresión. Esto quiere decir que esta es una buena forma de modelar el contenido de un archivo. El modelo generado luego de comprimir puede ser usado como modelo generativo para crear datos similares al que acabamos de comprimir. Simplemente tenemos que empezar en algún estado al azar y luego realizar un random-walk por el grafo de Markov que hemos aprendido. El resultado es que tanto PPMC como DMC pueden, luego de ser entrenados, usarse para generar textos, música u otros tipos de archivo de forma aleatoria. Cuanto mas parecidos sean estos archivos generados por el modelo a los archivos reales que podría generar una persona mas cerca está el algoritmo de "entender" de que se trata ese tipo de archivos. Estamos cerca.

4.8 La Familia LZ de Compresores

Los algoritmos de la familia LZ de compresores se basan en el principio de reemplazar secuencias repetidas en el archivo por punteros a la posición en la cual dichas secuencias fueron observadas previamente. El principio es muy simple y se basa en la suposición de que secuencias de caracteres previamente observadas pueden volver a ocurrir en el archivo; cuando esto ocurre podemos reemplazar una cantidad enorme de caracteres por un simple puntero a la posición previa de la secuencia y su longitud. Esta técnica, a veces llamada compresión por run-length fue popularizada a partir de 1977 por Lempel y Ziv con los algoritmos LZ77 y LZ78. Estos algoritmos dieron origen a lo que conocemos como la familia de compresores LZ que incluye una gran cantidad de algoritmos de compresión entre ellos LZ77,LZ78,LZW,LZSS, LZRW,LZFG,LZP, LZMA, Snappy y varios mas.

Antes de analizar el funcionamiento específico de cada uno de los algoritmos podemos generalizar el comportamiento de un compresor de la familia LZ de la siguiente forma:

1. Es necesario definir la longitud de un buffer o ventana que será en donde busquemos las repeticiones. Cuando mas grande el buffer mayor canti-

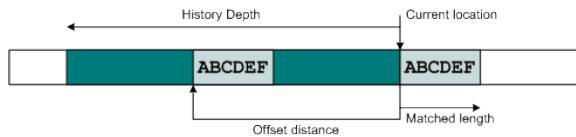


Figure 4.9: La familia LZ de compresores

dad de repeticiones podemos encontrar pero al mismo tiempo mas tiempo podemos tardar en procesar cada carácter.

2. De acuerdo a lo anterior suele ser necesario limitar el tamaño del buffer o usar indices para poder buscar rápidamente un string dentro del mismo.
3. La velocidad de la compresión en un LZ depende de la técnica usada para buscar las repeticiones en el buffer y la longitud del mismo.
4. El descomprimidor en todos los LZ es extremadamente rápido ya que solo necesita reemplazar las repeticiones por lo que haya en la posición del buffer indicado. En general en velocidad de descompresión los algoritmos LZ son imposibles de derrotar.
5. Cada repetición se reemplaza por símbolos que permiten al descomprimidor determinar la posición en el buffer y la longitud de la repetición

4.8.1 RLE

Nuestro primer algoritmo es extremadamente simple y sirve solo como una rápida introducción, el método llamado RLE (Run Length Encoding), tiene como único objetivo reemplazar secuencias de caracteres iguales repetidos en el archivo por una codificación más compacta. Cada vez que un carácter aparezca cuatro o más veces en el archivo lo vamos a reemplazar por los cuatro caracteres y un número en 8 bits indicando cuantas repeticiones mas agregar. De esta forma AAAAAAAA se transforma en AAAA4, AAAA se transforma en AAAA0 y cualquier otra cosa queda sin transformarse. Este método expande un byte por cada repetición de longitud 4, no gana ni pierde con repeticiones de longitud 5 y ahorra n-5 bytes por cada repetición de longitud "n".

ARCHIVO: ABCCCCCCCDDDDFFFFAAAAAAACD
COMPRIMIDO: ABCCCC2DDDDFFF0AAAA4CD

Descomprimir es muy simple solo tenemos que ir copiando los caracteres al archivo de salida pero si observamos cuatro caracteres iguales seguidos entonces el byte que leemos a continuación lo tomamos como un número indicando cuantas veces mas repetir el carácter anterior.

4.8.2 LZSS

LZSS es una versión mejorada de LZ77 el algoritmo que dio inicio a la familia de compresores LZ, al ser una versión mejorada de LZ77 en muchos casos se llama LZ77 a LZSS ya que la versión original no tiene un uso práctico. En este apunte lo vamos a llamar LZSS pero debemos mencionar que en otros lados el mismo algoritmo aparece bajo el nombre LZ77.

En LZSS vamos a usar un bit para distinguir repeticiones de no-repeticiones. Cada vez que estemos por comprimir un carácter del archivo vamos a buscar hacia atrás hasta una cierta cantidad máxima de caracteres llamada "ventana" en busca de repeticiones. Si encontramos una repetición entonces emitiremos un bit en 1 seguido de la posición de la repetición en la ventana y la longitud de la misma. Cuando no encontramos repeticiones vamos a emitir un bit en 0 y luego el carácter en cuestión en 8 bits es decir sin comprimir.

En principio entonces hay que fijar dos parámetros: la longitud de la ventana/buffer en la que buscaremos las repeticiones y la longitud mínima de un match, ya que puede ser que no tenga sentido representar un match de longitud 1 si ocupara más que el carácter sin comprimir.

Veamos un ejemplo usando longitud mínima de match = 2 y ventanas de 4 bytes.

ARCHIVO: ABCDADADAABDAA

MODELO: 0A 0B 0C 0D 0A 1(1,4) 0A 0B 1(3,3)

Observemos algunas particularidades. "0A" implica un bit en 0 y luego un carácter "A" es decir que en total usamos 9 bits. De la misma forma codificamos B,C,D y A. Luego de comprimir ABCDA tenemos en la ventana "BCDA" y el archivo "DADAABCDA" encontramos una repetición en la posición 1 del buffer, contamos a partir de 0 de izquierda a derecha. Y la longitud del match es 4 ya que "DADA" en el archivo matchea con "DADA" a partir de la posición 1 del buffer. Notemos que el buffer está limitado hacia atrás pero no hacia adelante, podríamos emitir un match de longitud 1Gb siempre y cuando esta longitud sea representable. Luego de este match emitimos una A, B,C y finalmente encontramos un match de longitud 2 en la posición 3 del buffer para los últimos tres caracteres del archivo.

Usando LZSS un archivo que solo tiene un carácter repetido n veces como por ejemplo AAAAAAAAAAAAAAAA se comprime en un literal y luego una cierta cantidad de matches de longitud máxima (excepto el último). Si la longitud máxima representable es mayor o igual a la longitud del archivo menos uno el archivo entero queda comprimido a un literal y un match. Notemos que las limitaciones en el tamaño de la máxima longitud representable y el tamaño de la ventana hacen que LZSS no sea un comprresor adecuado para aproximarnos a la complejidad de un string y por lo tanto no es usable en el cálculo de la distancia normalizada de compresión¹

¹LZMA sí lo es y pertenece a la misma familia que LZSS

Es conveniente tomarse unos minutos para entender la forma en que hemos convertido el archivo original en su representación.

Para comprimir lo único que nos resta definir es como representar los pares de tipo (posición, longitud). En LZSS esto se hace simplemente mediante códigos binarios de longitud fija. Como nuestra ventana tiene 4 bytes las posiciones posibles son 0,1,2 y 3 por lo que necesitamos 2 bits para representarlas, por lo tanto cada longitud será un binario de 2 bits. Las longitudes deben comenzar desde 2 y tenemos que fijar una longitud máxima, si por ejemplo decidimos usar 2 bits para las longitudes entonces podemos representar 4 longitudes y como el mínimo es 2 esto quiere decir que nuestras longitudes representables serán 2,3,4 y 5. Esto quiere decir que la cantidad de bits que usamos para la posición define la longitud de la ventana de búsqueda (o viceversa) y la longitud en bits que usemos para la longitud del match define el match máximo posible.

En nuestro caso el archivo quedaría representado de la siguiente forma:

MODELO: OA OB OC OD OA 1(1,4) OA OB 1(3,39)

ARCHIVO COMPRIMIDO: OA OB OC OD OA 10110 OA OB OC 11101 (73 bits)

Notar que "00" representa un match de longitud 2 porque es el mínimo fijado.

LZSS ilustra muy bien la diferencia entre modelar y codificar. El modelo se basa en aprovechar secuencias de caracteres previamente observadas y la codificación en este caso es un simple modelo binario.

La descompresión en LZSS es extremadamente rápida, cuando el descomprimidor encuentra un bit 0 emite los siguientes 8 bits como un carácter literal. Cuando el descomprimidor encuentra un bit 1 indicando una repetición lee los siguientes 2 bits para establecer la posición en la ventana, lee los siguientes 2 bits y suma la longitud mínima de match para determinar la longitud del match y luego simplemente copia esa cantidad de bits a partir de la posición indicada al archivo de salida. Dado que el descomprimidor no hace prácticamente nada es extremadamente veloz.

Parsing en LZSS

El mecanismo de parsing de LZSS es de tipo "lazy" es decir que siempre que encontramos un match lo reemplazamos por un offset y una longitud, curiosamente esto no siempre es óptimo ya que podría ser mejor ignorar un match para luego poder encontrar un match más largo. Consideremos el siguiente string:

curry urrent current

Cuando estemos por comprimir "current" vamos a buscar en el buffer y vamos a encontrar que matcheamos con "curr" por lo que vamos a emitir un match de longitud 4 y luego un match de longitud 3 para "ent". Si en cambio simplemente salteamos la "c" emitimos un literal "c" y luego tenemos un match

de longitud 6 (urrent). Dependiendo de la forma de codificar un literal y un match de 6 pueden ocupar menos espacio que un match de 4 y uno de 3.

La solución a este problema es, precisamente, analizar en cada paso la codificación de la posición actual y la siguiente y compararla con la codificación de un literal y lo que sigue eligiendo en cada caso la opción que menos espacio nos ocupe. A esto se lo llama **flexible parsing** y es una optimización posible en muchos de los algoritmos de la familia LZ. Notemos que para el descompresor no hace falta ningún cambio ya que simplemente va a descomprimir lo que el compresor haya decidido comprimir de la forma en que el mismo lo haya elegido.

4.8.3 DEFLATE (LzHuf)

DEFLATE es una evolución lógica de LZSS en donde mejoramos la forma de codificar. Es el algoritmo standard de PKZIP, WinZIP y similares tiene un nivel de compresión bastante aceptable y es muy rápido. Para empezar la modelización del archivo es idéntica a la de LZSS es decir que tenemos una ventana de una cierta longitud en donde buscamos matches.

ARCHIVO: ABCDADADAABCAA

MODELO: 0A 0B 0C 0D 0A 1(1,4) 0A 0B 0C 1(3,2)

La codificación del modelo es lo que cambia en DEFLATE. Para empezar vamos a tener un árbol de Huffman dinámico en donde conviven los 256 caracteres posibles y además todas las longitudes posibles desde LMIN hasta LMAX. Este árbol lo vamos a usar para codificar los caracteres literales cuando no encontramos un match y para las longitudes cuando encontramos un match. Cada vez que emitimos un carácter literal actualizamos el árbol incrementando su frecuencia y volvemos a usar el árbol para comprimir el carácter próximo. Cuando encontramos un match emitimos su longitud en base al código de la misma en el árbol y a continuación su posición en la ventana en una cierta cantidad de bits que se puede fijar mediante otro árbol de Huffman o de forma fija. Una vez que hicimos esto actualizamos la frecuencia de la longitud en el árbol y seguimos comprimiendo.

Notemos cuáles son las mejoras respecto de LZSS. En primer lugar ya no hace falta distinguir con un bit en 0 o 1 si se trata de un match o no. Cuando emitimos un carácter no hay match y cuando emitimos una longitud hay match. El descompresor sabe que si descomprime una longitud lo siguiente será una posición mientras que si descomprime un literal lo siguiente es o bien un literal o bien una longitud (mismo árbol). Una segunda ventaja es que podemos aprender cuáles son los caracteres literales más frecuentes en el archivo y usar menos bits para los mismos. Podemos también aprender cuáles son las longitudes más frecuentes en los matches y usar menos bits para estas longitudes que para las que son no tan comunes. Esto implica un importante ahorro de bits que termina generando un mejor nivel de compresión que LZSS.

4.8.4 LZW

LZW es una versión mejorada del algoritmo LZ78 publicado en 1978 por Lempel y Ziv, al igual que con LZ77 en muchos lados se llama LZ78 a LZW aunque sean algoritmos diferentes, la "W" en LZW es por Welch quien fue el que trabajó en las mejoras al algoritmo junto con Lempel y Ziv.

La idea principal de LZW es la misma que la de todos los algoritmos de la familia LZ: reemplazar secuencias previamente observadas por un código que las represente. LZW logra esto mediante la utilización de una tabla o diccionario en donde va almacenando las secuencias previamente observadas en el archivo, de esta forma cuando encuentra alguna secuencia que ya vio simplemente la reemplaza por el índice de la misma en el diccionario o tabla.

Inicialmente LZW comienza con una tabla de 512 posiciones en donde las primeras 256 entradas ya están llenas con las 256 combinaciones posibles de 8 bits, las restantes 256 posiciones están vacías. Cada posición de la tabla se representa mediante 9 bits. El algoritmo funciona entonces de la siguiente forma:

Algorithm 14: LZW

```

1 SYMBOL = readchar(FILE)
2 while not EOF do
3   | NEXT ← readchar(file)
4   | if SYMBOL + NEXT in TABLE then
5   |   | SYMBOL ← SYMBOL + NEXT
6   | else
7   |   | output code for SYMBOL
8   |   | if TABLE is not full then
9   |   |   | add SYMBOL+NEXT to table
10  |   |   | SYMBOL ← NEXT
11 output code for SYMBOL

```

Veamos un ejemplo de LZW en funcionamiento para el siguiente archivo:

ABCABCA

Vamos a hacer primero una narrativa del funcionamiento: empezamos leyendo "A" del archivo, como "A" ya está en la tabla en alguna de las primeras 256 posiciones entonces simplemente leemos otro byte: "AB", buscamos "AB" y no está en la tabla por lo tanto emitimos el código de "A", es decir su índice en la tabla que ocupa 9 bits y agregamos "AB" en la posición 256 de la tabla (numerando desde 0).

Nuestro símbolo actual es "B" lo buscamos en la tabla y existe, buscamos "BC" y no existe por lo tanto emitimos el código de "B" y agregamos "BC" en la tabla.

pos	chars
0	...
1	...
...	...
256	AB

Table 4.17: LZW luego de comprimir "A"

pos	chars
0	...
1	...
...	...
256	AB
257	BC

Table 4.18: LZW luego de comprimir "AB"

Nuestro símbolo actual es "C", está en la tabla, buscamos "CA" que no lo encontramos. Emitimos el código de "C" y agregamos "CA"

pos	chars
0	...
1	...
...	...
256	AB
257	BC
258	CA

Table 4.19: LZW luego de comprimir "ABC"

Tenemos ahora "A" que está en la tabla, leemos el próximo carácter del archivo y buscamos "AB" que también está en la tabla, leemos "C" y buscamos "ABC" que no está en la tabla. Por lo tanto emitimos el código de "AB" es decir 256 en 9 bits y agregamos ABC en la tabla.

Nuestro código actual es "C" que está en la tabla "CA" también y luego termina el archivo por lo que emitimos el código de CA (258) como nuestro último código. No hace falta agregar nada a la tabla ya que hemos terminado de comprimir el archivo, al igual que en cualquier compresor no tiene sentido actualizar el modelo si no vamos a tener que usarlo.

El archivo comprimido entonces queda:

ABCABCA

(A) (B) (C) (256) (258)

pos	chars
0	...
1	...
...	...
256	AB
257	BC
258	CA
259	ABC

Table 4.20: LZW luego de comprimir "ABCABC"

En total usamos 5 códigos de 9 bits cada uno es decir 45 bits para el archivo. Para completar a un múltiplo de 8 podemos agregar bits al azar ya que nunca serán suficientes para que el descompresor genere símbolos equivocados ya que los códigos son de 9 bits (o mas) y el padding es a lo sumo 7 bits.

Tenemos que analizar que sucede si la tabla de LZW se llena lo cual va a ocurrir relativamente rápido ya que solo tenemos lugar para 256 strings, una vez que la tabla de 9 bits se llena el compresor simplemente duplica la tabla y luego procede a emitir códigos de 10 bits en lugar de 9, este proceso se repite tantas veces como sea necesario.

Hay algunos detalles a tener en cuenta aquí ya que no está del todo claro en la especificación si el compresor tiene que emitir 10 bits cuando se llena la tabla o un símbolo mas adelante. Consideremos la siguiente situación para entender el problema:

pos	chars
0	...
1	...
...	...
256	AB
257	BC
258	CA
259	ABC
...	...
510	...

Table 4.21: LZW a punto de llenar la tabla

Leemos caracteres del archivo hasta encontrar un string que no está en la tabla. Por ejemplo "ABCAB" (podría ser cualquier otra cosa). Tenemos entonces que emitir el código de ABCA que sabemos que está en la tabla y luego agregar ABCAB a la tabla:

Al haberse llenado la tabla sabemos que el próximo string a guardar no va a entrar en la misma pero al mismo tiempo sabemos que el próximo código a

pos	chars
0	...
1	...
...	...
256	AB
257	BC
258	CA
259	ABC
...	...
510	...
511	ABCAB

Table 4.22: LZW con la tabla de 9 bits llena

emitir será de 9 bits ya que tiene que ser si o si algo que exista en la tabla actual... Tenemos entonces dos opciones:

1. Duplicar la tabla y emitir el próximo código en 10 bits
2. Emitir el próximo código en 9 bits y luego duplicar la tabla para agregar el símbolo nuevo

No hay un standard por lo que lamentablemente muchos compresores y descompresores LZW necesitan algún tipo de flag para indicar cuál de estas dos políticas han usado. El formato PDF usa LZW y admite ambas políticas de actualización del diccionario mediante un flag.

El Descompresor LZW

La descompresión en LZW es muy similar y sigue el camino inverso al compresor, sin embargo es un tema que no resulta sencillo ya que LZW es el único de todos los algoritmos de compresión en el cual el compresor y el descompresor no van sincronizados perfectamente, esta característica es especialmente molesta para programar compresores y descompresores LZW y también para entender su funcionamiento.

```

LEEMOS (A) TABLA: 256=A? OUT:A
LEEMOS (B) TABLA: 256=AB, 257=B? OUT:B
LEEMOS (C) TABLA: 256=AB,257=BC,258=C? OUT:C
LEEMOS (256) TABLA: 256=AB,257=BC,258=CA, 259=AB? OUT: AB
LEEMOS (258) TABLA: 256=AB,257=BC,258=CA, 259=ABC, 260=CA? OUT:CA

```

Veamos el comportamiento del descompresor paso a paso: leemos inicialmente un código de 9 bits que corresponde a "A" por lo tanto podemos emitir "A" al archivo de salida y además sabemos que la entrada 256 de la tabla es "A" seguido de algo mas es decir "A?". Luego leemos el código de 9 bits que

corresponde a la "B" podemos emitir "B" y completar ahora 256=AB y agregar 257=B? en la tabla. Luego leemos "C" así que completamos 257=BC y agregamos 258=C? además de emitir C. Luego leemos el código 256 que es "AB" así que completamos 258=CA, notemos que solo nos interesa el primer carácter, y agregamos 259=AB? además de emitir AB. Finalmente leemos 258 y emitimos "CA", podemos si queremos completar 259=ABC y agregar 260=CA? pero esto ya no lo vamos a usar.

Debería ser evidente ahora la diferencia en sincronización entre el compresor y el descompresor. El compresor agrega a la tabla un nuevo símbolo luego de comprimir un carácter, el descompresor sin embargo solo sabe cuál es el símbolo agregado por el compresor al leer el siguiente. Esto es particularmente perturbador pero no impide que se pueda comprimir y descomprimir pero nos lleva a un caso muy curioso con el cual hay que tener un terrible cuidado.

Los Dragones de LZW

Sea el siguiente string:

ABABABAA

Vamos a comprimirlo usando LZW tal y como ya sabemos.

Empezamos leyendo "A" que está en la tabla, agregamos "AB" que no está por lo tanto emitimos el código de 9 bits de "A" y agregamos 256=AB, ahora tenemos "B" que está en la tabla, agregamos "BA" que no está, por lo que emitimos "B" en 9 bits y 257=BA, tenemos ahora "A" que está en la tabla y AB también está en la tabla por lo que leemos otro carácter y tenemos "ABA" que no está en la tabla. Emitimos el código de AB (256) y agregamos 258=ABA. Tenemos ahora "A", leemos "AB", leemos "ABA", "ABAA" que no está en la tabla por lo que emitimos el código de ABA (258) y agregamos 259=ABAA. Finalmente emitimos el código de la última A.

ABABABAA

(A) (B) (256) (258) (A)

Veamos el funcionamiento del descompresor

```
LEEMOS (A) TABLA: 256=A? OUT:A
LEEMOS (B) TABLA: 256=AB, 257=B? OUT: B
LEEMOS (256) TABLA: 256=AB, 257=BA, 258=AB? OUT: AB
LEEMOS (258) ???
```

Se da el terrible caso en el cual el descompresor lee un código que no está en la tabla. Sin embargo a no desesperar porque el descompresor puede deducir lo que ha sucedido en el compresor y gracias a esto reconstruir el código faltante. El descompresor sabe que el código 258 empieza con "AB" y sabe que el próximo carácter que completa el código es el primero del que sigue, que es también el 258 por lo tanto el último carácter de 258 es el primer carácter de 258 y 258 es "ABA".

```

LEEMOS (A) TABLA: 256=A? OUT:A
LEEMOS (B) TABLA: 256=AB, 257=B? OUT: B
LEEMOS (256) TABLA: 256=AB, 257=BA, 258=AB? OUT: AB
LEEMOS (258) TABLA: 256=AB, 257=BA, 258=ABA, 259=ABA? OUT:ABA
LEEMPS (A) TABLA: 256=AB, 257=BA, 258=ABA, 259=ABAA OUT:A

```

De esta forma podemos descomprimir todo el archivo.

Mecanismo de Clearing

Hemos visto que en LZW a medida que agregamos strings en el diccionario emitimos cada vez mas bits, llega un punto en el cual el nivel del compresor puede ser malo, esto se debe a que tiene en la tabla muchos strings que ya no vuelven a ocurrir en el archivo y por lo tanto está generando códigos mucho mas grandes de lo necesario. Cuando el compresor detecta que el nivel de compresión (bits x byte) no es bueno puede usar un símbolo especial para avisarle al descompresor que tiene que hacer una purga de la tabla. Para esto podemos, por ejemplo, dedicar los símbolos 256,257,etc empezando a guardar los strings a partir del 258 o mas adelante, todo depende de cuantos "comandos" queramos tener a nuestra disposición en el compresor. Algunos de estos comandos pueden ser por ejemplo:

1. Eliminar todas las tablas, volver a 9 bits
2. Eliminar de las tablas los símbolos menos usados para pasar a 1 bit menos
3. Eliminar de las tablas todos los símbolos que se emitieron menos de x veces (x sigue a continuación en "m" bits)

El compresor solo necesita llevar las estadísticas necesarias y cuando cree conveniente puede emitir alguno de estos códigos especiales. El descompresor al leer alguno de estos códigos especiales replica la acción tomada por el compresor.

Suboptimalidad de la codificación en LZW

Como hemos mencionado que comprimir es modelar y codificar debemos notar que en LZW la codificación no es óptima, esto es muy fácil de demostrar, al empezar con códigos de 9 bits podemos emitir 512 símbolos diferentes, de los cuales solo 256 son posibles, luego solo 257 son posibles y así sucesivamente, es decir que existen códigos que el compresor nunca puede emitir. Por ejemplo el segundo código emitido puede ser a lo sumo 256 pero nunca 257 o superior. El descompresor también sabe esto pero sin embargo no hay forma de solucionarlo, estos códigos imposibles hacen a la ineficiencia de LZW ya que no cualquier archivo es algo que se pueda descomprimir con LZW.

Una forma de solucionar esto es usando compresión aritmética, en donde al principio tenemos 256 elementos en la tabla, todos equiprobables y luego al agregar el primer string al diccionario pasamos a tener 257 símbolos y así sucesivamente, la codificación aritmética como sabemos solo necesita $-\log_2(1/257) =$

8.00562454 bits que es mucho menos que los 9 bits que emitiríamos en la versión tradicional de LZW.

Complejidad LZ

Es posible usar LZ78 para estimar la complejidad de un string [Lempel-Ziv-C] desde el punto de vista de este compresor. La complejidad se define simplemente como la cantidad de patrones diferentes que ocurren en el string. Por ejemplo si tenemos el string 010101011 empezamos con el primer bit 0 y lo agregamos a la tabla. El segundo bit 1 no está en la tabla por lo que lo agregamos. Luego leemos 0 que ya fue visto así que leemos otro bit: 01 no está en la tabla y lo agregamos. De esta forma continuamos y nos queda algo de tipo: 0|1|01|010|11| con lo cual estimaríamos que la complejidad LZ de este string es 5.

Es evidente que en un string o archivo en donde hay muchos patrones repetidos vamos a tener una complejidad baja mientras que en un string o archivo completamente aleatorio eventualmente tendríamos una gran cantidad de patrones diferentes.

Es posible usar esto para calcular la entropía de acuerdo a un compresor LZ simplemente calculando la probabilidad de cada patrón que es la cantidad de veces que el mismo ocurre en el archivo dividido la cantidad total de patrones encontrados. Veamos qué pasa con un archivo aleatorio:

0|1|01|10|11|00|001|100|010|011|101 Como podemos ver la frecuencia de todos los patrones es 1 y la cantidad total de patrones depende de la longitud del archivo. Vamos a tener siempre 2 patrones de longitud 1, 4 patrones de longitud 2, 8 patrones de longitud 3 etc. Eventualmente podemos probar que para un archivo random la entropía de acuerdo a la complejidad LZ es igual a la entropía simplemente calculando la probabilidad de cada bit es decir igual a la longitud del archivo mismo. Dejamos esto como un sencillo ejercicio.

4.8.5 Snappy

Snappy es una variante de LZ optimizada para comprimir y descomprimir con gran velocidad, esto se logra en sacrificio de la compresión. El algoritmo fue creado por Google para su producto BigTable y sigue siendo usado en muchos productos de la empresa para almacenar información comprimida sin perder la capacidad de recuperarla velozmente.

En Snappy el primer byte de cada bloque indica en sus primeros 2 bits el tipo de dato que sigue a continuación

```
00 = literales
01 = 1 byte match
10 = 2 byte match
```

Literales

Cuando se indican literales (primeros dos bits =00) los siguientes 6 bits se usan para la longitud de los mismos. De la siguiente forma:

```

0 a 59 : Cantidad de literales 1 a 60
60: La cantidad de literales se indica en el próximo byte.(61 a 61+256)
61: La cantidad de literales se indica en los próximos dos bytes.
62: La cantidad de literales se indica en los próximos tres bytes.
63: La cantidad de literales se indica en los próximos cuatro bytes.

```

Es decir que algo de tipo 00 000100 indica que se trata de literales (00) y la longitud = 4 indica que se trata de 5 literales sin comprimir por ejemplo "ABCDE". Si los literales son 200 entonces hay que indicarlo mediante 00 111100 10001011. Los dos primeros bits indican que es un literal (00) los siguientes 6 bits son 60 indicando que la longitud de los literales se indica en el próximo byte y los siguientes 8 bits son 139 indicando que la longitud es 200 (61+139).

Por lo tanto Snappy puede representar tiras de literales sin comprimir de 1 a 60 bytes usando n+1 bytes y puede llegar hasta varios gigas de literales ocupando n+5 bytes (1 de control y 4 para la longitud)

Matches

Los matches de longitud 4 a 11 con offsets 1 a 2047 se almacenan usando 1 byte extra. La longitud del match menos 4 se almacena en los primeros 3 bits de los 6 bits que sobran en el byte de control. Los restantes 3 bits y los 8 bits del byte siguiente se usan para almacenar el offset.

Ejemplo:

ARCHIVO: abababa

```

000001 00 (literal of length 2)
01100001 (literal 'a')
01100010 (literal 'b')
000 001 01 (high bits of offset, match of length 5)
00000010 (low 8 bits of offset)

```

Los matches de longitud 1 a 64 con offsets 1 a 65535 se almacenan usando 2 bytes extras. La longitud menos 1 se almacena en los 6 bits del byte de control. El offset se almacena en los siguientes 2 bytes. Los matches de mas de 64 bytes de longitud se almacenan como varios matches de 64 bytes.

La secuencia entera de matches y literales se almacena precedida por la longitud del archivo original hasta $2^{32} - 1$ en formato de longitud variable. De la siguiente forma: el primer bit de cada byte indica si el número termina en dicho byte (0) o continua en el siguiente. Los restantes 7 bits tienen el número. De esta forma en 1 byte podemos almacenar desde 1 a 2^7 y en n bytes podemos almacenar hasta 2^{7n} .

Para buscar matches Snappy usa una tabla de hash hasheando los 4 bytes que siguen a comprimir. Como una optimización para archivos que son difíciles de comprimir el compresor se fija si ha habido 32 fallos seguidos de encontrar una repetición en cuyo caso empieza a chequear cada 2 bytes, luego de 32 fallos

cada 3 bytes y así sucesivamente. Cuando encuentra un match vuelve a chequear byte por byte.

4.8.6 LZMA

LZMA es el algoritmo de la familia LZ que mejor nivel de compresión logra. Es el algoritmo usado por 7-zip y está a la altura de los mejores compresores del mercado lo cual nos lleva a pensar que el modelo de la familia LZ puede ser igual de óptimo que el modelo probabilístico de la familia PPM aunque sean cosas completamente diferentes. En definitiva es posible pensar que varios modelos diferentes nos lleven a tender a la complejidad de Kolmogorov del string.

LZMA es un formato complejo por lo que vamos a describir cuáles son sus principales características y "trucos" con respecto a otros LZs sin entrar en detalles sobre como se implementan.

Tipos de Códigos

En cada paso LZMA puede generar 7 códigos diferentes, esto es mas que lo normal en LZSS en donde solo hay dos códigos posibles (literal o match), la siguiente tabla muestra cuáles son los códigos que puede emitir un compresor LZMA pero hay que tener en cuenta que los 1s y 0s en la tabla no son bits sino símbolos que luego hay que codificar y que pueden ocupar mas o menos de un bit cada uno. A efectos prácticos podríamos llamarlos "A" y "B" pero entre dos opciones confusas es confuso elegir cual es menos confusa.

code	name
0 + char	LITERAL
1 + 0 + len + dist	MATCH
1 + 1 + 0 + 0	SHORTREP
1 + 1 + 0 + 1 + len	LONGREP[0]
1 + 1 + 1 + 0 + len	LONGREP[1]
1 + 1 + 1 + 1 + 0 + len	LONGREP[2]
1 + 1 + 1 + 1 + 1 + len	LONGREP[3]

Table 4.23: Códigos de LZMA

- Un LITERAL representa simplemente un byte literal al igual que en LZSS.
- MATCH es un match en el estilo de LZSS indicando longitud (len) y distancia en el buffer al mismo.
- SHORTREP es un match de longitud 1 cuya distancia es igual a la distancia anterior emitida
- LONGREP[i] es un match cuya longitud se indica y cuya distancia es igual a la i-ésima distancia anteriormente emitida, es decir la última para LONGREP[0], la ante-última para LONGREP1 y así sucesivamente.

De esta forma sabemos como modela LZMA a cualquier archivo. La figura 4.10 muestra el modelo de LZMA para el archivo book1, los literales aparecen en gris, los matches comunes estilo LZSS en verde. Las SHORTREP aparecen en azul. LONGREP0 en blanco y LONGREP1-3 en rojo.

```

Detailed*bore*some*reduced*resemblance*to*features*here.*  

<P*58>*  

The*thin*grasses,*more*or*less*coating*the*hill,*were*  

touched*by*the*wind*in*breezes*of*differing*powers,*and*  

almost*of*differing*natures*--*one*rubbing*the*blades*  

heavily,*another*raking*them*piercingly,*another*brushing*  

them*like*a*soft*broom.*The*instinctive*act*of*human+*  

kind*was*to*stand*and*listen,*and*learn*how*the*trees*  

so*much*other*in*the*regular*antiphonies*of*a*cathedral*  

choir;*how*hedges*and*other*shapes*to*leeward*them*  

caught*the*note,*lowering*it*to*the*tenderest*note;*and*  

now*the*hurrying*gust*then*plunged*into*the*south,*to*  

be*heard*no*more.*  

The*sky*was*clear*--*remarkably*clear*--*and*the*  

twinkling*of*all*the*stars*seemed*to*be*about*throbs*of*  

one*body,*timed*by*a*common*pulse.*The*North*Star*  

was*directly*in*the*wind*s*eye,*and*since*evening*the*  

Bear*had*swung*round*it*outwardly*to*the*east,*till*he*  

was*now*at*a*right*angle*with*the*meridian.*A*  

difference*of*colour*in*the*stars*--*oftener*read*of*than*  

seen*in*England*was*really*perceptible*here.*The*  

sovereign*brilliancy*of*Sirius*pierced*the*eye*with*a*steely*  

glitter,*the*star*called*Capella*was*yellow,*Aldebaran*and*  

Betelgeux*shone*with*a*fiery*red.*  

To*persons*standing*alone*on*a*hill*during*a*clear*  

midnight*such*as*this,*the*call*of*the*world*eastward*is*  

almost*a*palpable*movement.*The*sensation*may*be*  

caused*by*the*panoramic*glide*of*the*stars*past*earthly*  

objects,*which*is*perceptible*in*a*few*minutes*of*still+*

```

Figure 4.10: Modelo LZMA para el archivo book1

Codificación Aritmética

Conociendo el modelo solo queda codificarlo y LZMA usa compresión aritmética para codificar todos sus símbolos. Es decir que comenzamos con un modelo en el cual solo hay dos opciones: LITERAL u OTRACOSA, según lo que vayamos a codificar entonces elegimos el intervalo correspondiente. Por ejemplo si elegimos "OTRACOSA" entonces en el siguiente paso el próximo modelo distingue un MATCH de OTRACOSA. Y así sucesivamente de acuerdo a la tabla de códigos

posibles que podemos emitir. Los literales se comprimen usando compresión aritmética adaptativa. Las longitudes se codifican de la siguiente forma (una vez mas 1 y 0 no son bits)

Len	Desc
0 + 3 bits	Longitudes 2 a 9
1+0+3 bits	Longitudes 10 a 17
1+1+8 bits	Longitudes 18 a 273

Table 4.24: Longitudes de LZMA

La codificación de las distancias (offsets) se codifican con un esquema bastante particular que preferimos no describir en este apunte para no confundir.

Tamaño de la Ventana y Optimal Parsing

LZMA usa un buffer de 4Gb (!) y hashing para encontrar los matches, además usa optimal parsing analizando las diferentes opciones que tiene el compresor en cada paso y luego de "ver" algunos pasos hacia adelante selecciona la estrategia de compresión mas conveniente.

Uso de Exclusión

LZMA usa una forma de exclusión para los literales. Ya que sabe que el carácter que seguía a un match previamente emitido no puede ser el literal que viene a continuación pues hubiese formado parte del match. Por lo tanto puede eliminarlo de la tabla de literales posibles mejorando un poco el nivel de compresión.

LZMA es el mas avanzado de los algoritmos de la familia LZ y logra niveles de compresión que están a la altura de los mejores compresores actuales y todo esto simplemente en base a un modelo de aprovechar secuencias repetidas del archivo para reemplazarlas por códigos.

4.9 Block Sorting

El algoritmo de Block Sorting es una de las historias mas sorprendentes del mundo de la compresión de datos, el modelo que usa Block Sorting para comprimir es extremadamente original y en cierta forma solo puede justificarse a partir de su éxito. Para entender el funcionamiento de Block Sorting vamos a empezar primero con otro algoritmo relacionado llamado MTF.

4.9.1 Archivos localizados y MTF

Vamos a decir que un archivo está **localizado** cuando en determinadas zonas del mismo hay preponderancia de un cierto conjunto de caracteres, por ejemplo algo de la forma:

ADCBABDCCAFGFHJHGHHFJF

Es un archivo fuertemente localizado ya que en la primera parte solo hay A,B,C y D y en la segunda parte F,G,H y J. Sin embargo veamos que esto no implica que haya secuencias repetidas, un archivo localizado es una generalización de las secuencias repetidas, en cierta forma cualquier archivo con secuencias repetidas está localizado pero no cualquier archivo localizado tiene secuencias repetidas.

Cuando un archivo está localizado el algoritmo MTF (Move to Front) nos permite representarlo mediante un modelo que luego podemos codificar muy eficientemente. El algoritmo es muy simple: cuenta con un vector de 256 posiciones que comienza con los 256 caracteres posibles, por cada byte a procesar emitimos el índice del mismo en el vector y luego lo movemos al principio del mismo. Veamos un ejemplo en donde usamos el set de caracteres ABCD para no tener que usar los 256 bytes posibles:

ARCHIVO: ABBAABACDDCCDC

Leemos	Vector	Emitimos
A	ABCD	0
B	ABCD	1
B	BACD	0
A	BACD	1
A	ABCD	0
A	ABCD	0
B	ABCD	1
A	BACD	1
C	ABCD	2
D	CABD	3
D	DCAB	0
C	DCAB	1
C	CDAB	0
D	CDAB	1
C	DCAB	1

Table 4.25: Ejemplo MTF

Observemos que MTF cambia la entropía del archivo, pasamos de $H[5/15;3/15;4/15;3/15]=1.9656$ a $H[6/15;7/15;1/15;1/15]=1.5628$. Siempre que el archivo esté localizado MTF va a disminuir su entropía. En el resultado de MTF predominan los caracteres bajos (0,1,etc) sobre los demás caracteres esto lo podemos aprovechar comprimiendo la salida de MTF usando Huffman o compresión aritmética. Como la entropía es menor sabemos que el nivel de compresión va a ser mejor.

La descompresión de MTF es trivial ya que simplemente leemos números de 8 bits que tomamos como índices en el vector emitiendo lo que encontramos en el mismo y moviendo dicho carácter al principio del vector.

4.9.2 La Transformación de Burrows y Wheeler

La transformación de Burrows y Wheeler [Burrows-Wheeler] permite tomar un archivo común (no random) y convertirlo en un archivo localizado. Esta transformación funciona de la siguiente forma: En primer lugar dividimos el archivo en bloques de un cierto tamaño, por ejemplo 900Kb y construimos todas las rotaciones del bloque. Esto lo vamos a ilustrar con un ejemplo pero en la práctica no implica hacer nada salvo guardar un puntero a cada byte del bloque, si pensamos en el bloque como si fuera una cola circular entonces cada puntero representa una rotación del mismo.

```

0 PANQUEQUE
1 ANQUEQUEP
2 NQUEQUEPA
3 QUEQUEPAN
4 UEQUEPANQ
5 EQUEPANQU
6 QUEPANQUE
7 UEPANQUEQ
8 EPANQUEQU

```

A continuación hacemos un sort de los punteros, lo cual equivale a ordenar todas las rotaciones del bloque.

```

0 ANQUEQUEP
1 EPANQUEQU
2 EQUEPANQU
3 NQUEQUEPA
4 PANQUEQUE
5 QUEPANQUE
6 QUEQUEPAN
7 UEPANQUEQ
8 UEQUEPANQ

```

El resultado de la transformación es la última columna de la matriz ordenada junto con un índice que indica en qué número de rotación queda el archivo original. Es decir:

PANQUEQUE => PUUAEENQQ + 4

Notemos como el archivo, aun en un caso tan pequeño ha quedado localizado. Todas las U juntas, las E, las Q. Esto se puede explicar si pensamos que la última columna representa el carácter anterior al que está en la primer columna. Cuando ordenamos las rotaciones al quedar todas las "U" juntas en la primer columna nos quedan todas las "Q" juntas en la última columna. Eventualmente mezcladas con otros caracteres que suelen encontrarse antes de "U" pero eso no quita a la definición de archivo localizado. En cierta forma la transformación de BW se basa en la intuición de que no todos los caracteres pueden aparecer

antes que otro para construir un archivo en donde dichos caracteres quedan localizados dentro de una misma zona del archivo.

Para revertir la transformación de Burrows y Wheeler lo que haremos es en primer lugar ordenar los caracteres:

```
0 P
1 U
2 U
3 A
4 E
5 E
6 N
7 Q
8 Q
```

```
(sort)
0A 3
1 E 4
2 E 5
3 N 6
4 P 0
5 Q 7
6 Q 8
7 U 1
8 U 2
```

Una vez que llegamos a esto empezamos por el índice que es "4" y vemos una P que está asociada con un 0 que es "A" que está asociada con un 3 que es "N" y así sucesivamente hasta descomprimir "PANQUEQUE".

Una vez realizada la transformación de BW tenemos un archivo localizado por lo que podemos aplicarle MTF y finalmente codificar con Huffman o compresión aritmética. Este "stack" es capaz de comprimir con un excelente nivel de compresión y se basa en principios completamente diferentes a los que vimos en PPMC o LZMA. El compresor BZIP2 es una implementación directa del algoritmo de Block Sorting, MTF y Huffman para codificar.

4.10 PAQ

PAQ es el algoritmo de compresión que mejor nivel de compresión alcanza para archivos en general, es decir en promedio para cualquier archivo (no random). Podemos pensar que PAQ es entonces el algoritmo cuyo modelo mejor logra "entender" a un archivo y queremos saber cuál es dicho modelo. La respuesta la pueden imaginar: el modelo de PAQ es el ensamblaje de varios modelos diferentes.

PAQ es un compresor aritmético que funciona bit a bit, es decir que por cada bit a comprimir del archivo PAQ estima la probabilidad de que dicho bit sea 1 o 0 y subdivide el intervalo actual de acuerdo a dichas probabilidades

quedándose con el intervalo que le corresponde al bit leído del archivo. Cuanto mejor sea capaz PAQ de estimar las probabilidades menos bits va a necesitar para representar la precisión necesaria para el siguiente intervalo y por lo tanto el archivo comprimido va a ser más pequeño.

Para llegar a la probabilidad del 1 y el 0 PAQ va a combinar la predicción de varios modelos. Mas adelante veremos cuáles son estos modelos supongamos ahora que tenemos "n" modelos y que cada uno de ellos calcula la probabilidad de que el próximo bit sea un 1. Lo que PAQ necesita es encontrar pesos de forma tal que la probabilidad final de que el bit sea 1 surja del promedio ponderado de las probabilidades de todos los modelos utilizados. Estos pesos en PAQ surgen de una red neuronal. El funcionamiento es en realidad muy simple, todos los bits anteriores y las predicciones de los modelos sirven como set de entrenamiento.

M0	M1	M2	M3	M4	OBS
0.3	0.2	0.8	0.4	0.05	0
0.7	0.3	0.9	0.75	0.76	1
0.65	0.2	0.83	0.69	0.77	1
0.03	0.1	0.3	0.65	0.22	0

Table 4.26: Ejemplo PAQ

La figura 3.23 nos muestra un ejemplo del set de entrenamiento que PAQ se auto-construye al ir comprimiendo el archivo. En base a las predicciones realizadas por los modelos y el bit observado realmente en el archivo (OBS) PAQ puede entrenar una red neuronal de forma tal que dadas las probabilidades de los modelos calcule la probabilidad de que el bit sea 1 de forma tal de ajustar de la mejor forma posible al set de entrenamiento es decir a lo que observamos realmente en el archivo. Esta forma de mezclar modelos se llama "context mixing" y es la clave de un algoritmo de compresión exitoso ya que en diferentes circunstancias uno u otro modelo puede funcionar mejor, a medida que PAQ va comprimiendo puede aprender cuáles son los modelos que mejor ajustan al archivo y darles el peso que corresponde para predecir mejor. Estos pesos van cambiando dinámicamente es decir que a medida que avanzamos en el archivo podemos ir descubriendo que un modelo empieza a ser mejor que otro, esto se logra automáticamente al ir ajustando la red neuronal con nuevas observaciones.

4.10.1 Modelos de PAQ

PAQ utiliza una combinación de modelos, no queremos entrar en grandes detalles sobre el funcionamiento de cada uno de estos modelos pero a grandes rasgos los modelos más usados son los siguientes:

1. Un modelo basado en los "n" (hasta 8) caracteres anteriores y los "m" bits (hasta 7) anteriores al bit actual. Este es un modelo idéntico al que usaría PPMC, en base al contexto calculamos la probabilidad de un bit 0 o 1, no hace falta usar escapes porque solo puede haber dos símbolos en

cada contexto (0 o 1) lo cual hace que podamos tener todo en memoria y que no sea tan lento aprender ya que empezamos de $1/2 \ 1/2$ para cada bit. Notemos que procesar bit a bit soluciona el problema de frecuencia cero automáticamente!.

2. Un modelo basado en repeticiones de strings. PAQ busca en un buffer de 4Mb contextos mayores a 8 bytes que matcheen el contexto actual, cuando estos contextos existen PAQ toma nota de cuál es el bit siguiente en cada caso y en base a eso calcula la probabilidad de que el próximo bit sea 1 o 0.
3. Un modelo basado en palabras completas. Aquí PAQ toma como contexto una o dos palabras y "m" bits para predecir la probabilidad de que el próximo bit sea un 1 o 0, usando puntuación y/o espacios en blanco para separar palabras.
4. Un modelo que predice estructuras repetitivas como por ejemplo tablas.

4.10.2 SSE

Una vez que ha calculado la probabilidad de que el próximo bit sea 1 o 0 PAQ usa SSE (secondary symbol estimation) para refinar su predicción, lo que hacemos aquí es tomar "n" bits y la probabilidad discretizada en 64 intervalos como contexto y en base a eso verificar la probabilidad del 1 y el 0 ajustando mediante una fórmula variable la probabilidad conjunta de los modelos y esta estimación. Es decir que por un lado PAQ tiene una estimación por cada modelo, todas esas estimaciones y los bits realmente observados se usan para entrenar una red neuronal que calcula la probabilidad de que el próximo bit sea un 1. Con esta probabilidad y el contexto PAQ se fija cuantas veces ha aparecido realmente un 0 y cuantas veces un 1. Por ejemplo sea $P=0.33$ si los bits anteriores son 01000 y la probabilidad está entre 0.2 y 0.4 el bit próximo fue un 0 9 de 10 veces entonces en lugar de predecir $P=0.33$ podemos ajustar P a un número mas bajo ya que estamos mas seguros de que el próximo bit va a ser un cero.

Mediante la combinación de varios modelos usando una red neuronal y SSE PAQ es capaz de construir un modelo que se adapta muy elásticamente al contenido del archivo, dándole peso a un modelo u otro según el contenido del mismo de forma dinámica lo cual le permite comprimir realmente muy bien. Hasta el próximo avance en compresión de datos PAQ representa la forma mas avanzada que tenemos de estimar la complejidad de Kolmogorov de un archivo y es el compresor ideal a usar para calcular la distancia normalizada de compresión cuando querremos estimar la distancia existente entre dos archivos. Esto tiene muchas aplicaciones.

Chapter 5

Hashing

The secret impresses no one. The trick you use it for is everything.

Alfred Borden, The Prestige

5.1 Introducción

El estudio de este tema debe plantearse desde varios puntos de vista distintos. Por un lado estudiaremos funciones de hashing, funciones que nos permiten convertir un dato de cualquier tipo en un número lo cual puede eventualmente resultar de gran utilidad. Por otro lado estudiaremos que usando funciones de hashing podemos acceder de forma muy rápida a una tabla o vector en memoria, de esta forma una función de hashing sirve para generar una estructura de datos que denominamos "tabla de hash" y que es una de las estructuras de datos mas usadas debido a su eficiencia. Finalmente veremos que las funciones de hashing están relacionadas también con la posibilidad de representar datos de forma compacta.

A modo de empezar vamos a ver algunas funciones de hashing muy básicas que no se usan en la práctica y algunas formas rudimentarias de implementar una tabla de hash usando estas funciones.

5.1.1 Funciones de Hashing Elementales

En su mas básica definición una función de hashing debe ser capaz de recibir como input un cierto dato cualquiera y devolver un número acotado a un cierto intervalo. Al conjunto de claves posibles (input) lo denominaremos "espacio de claves" y al conjunto de números que la función de hashing puede devolver lo llamaremos "espacio de direcciones".

Por ejemplo consideremos una función de hashing que transforma un número de patente en formato (AAA999) en un número entre 0 y 50000. Como todas

las patentes son posibles el espacio de claves es de $26^3 * 1000 = 17576000$ y el espacio de direcciones es evidentemente 50000.

Analicemos ahora los dos casos posibles:

El espacio de claves es mayor al espacio de direcciones

Este suele ser el caso mas frecuente y es una buen motivo para justificar la necesidad de una función de hashing. Supongamos que queremos almacenar 1 millón de strings de longitud 140 en memoria. En total existen 26^{140} o mas strings posibles por lo que de alguna forma tenemos que "comprimirlos" a un número entre 0 y 999999, esto se logra precisamente con una función de hashing.

El espacio de direcciones es mayor al espacio de claves

Este caso es menos frecuente pero también podría llegar a darse, por ejemplo por razones de eficiencia podríamos querer almacenar 25000 valores en una tabla con capacidad para 1 millón de datos, está bien pensar que esto, en principio, parece no tener mucho sentido pero ocurre que en ciertos algoritmos es algo deseable por lo que dejaremos planteado que es un escenario plausible.

Para construir una función de hashing necesitamos de alguna forma convertir un dato cualquiera en un número entre 0 y m, una forma de lograr esto es convertir el dato en un número z y luego aplicar la función módulo: $z \bmod m$. Cuando los datos son numéricos esto puede hacerse directamente. Si los datos son, por ejemplo, strings de longitud fija, podemos convertir cada carácter en un código ASCII o tomar el binario que corresponde cada 2 caracteres y sumar para obtener un número. Si los datos fueran de longitud variable podríamos ir tomando de a 2 bytes en binario e ir sumando y calculando el modulo m, esto evita el overflow y permite reducir una tira de bits de cualquier longitud a un número entre 0 y m que es lo que queremos. Este tipo de funciones de hashing se denominan "fold and add", son muy fáciles de entender pero desafortunadamente no son muy eficientes, suelen generar muchas mas colisiones que las deseadas y además no son muy rápidas. Sin embargo son útiles para entender que es lo que una función de hashing debe hacer.

5.1.2 Tablas de Hash Elementales

Supongamos que contamos con alguna función de hashing para un cierto tipo de datos. Una aplicación directa de esta función es construir una estructura de datos muy simple y eficiente que se denomina "Tabla de hash". Una tabla de hash no es mas que un vector de tamaño m en el cual almacenamos los datos en la posición indicada por aplicarle al dato la función de hashing. De esta forma si queremos almacenar "Roma" en nuestra tabla de hash calculamos $h("Roma")$ y si el resultado nos da por ejemplo 19 almacenamos el string en la posición 19 de la tabla. De la misma forma usaremos la función de hashing para buscar datos en nuestra tabla sin tener que recorrerla ni mantenerla ordenada.

En un caso ideal una tabla de hash permite almacenar y recuperar datos en $O(1)$ lo cual es, evidentemente, muy bueno. Sin embargo esto requiere que la función de hashing sea perfecta, es decir que no produzca colisiones. Una colisión se produce cuando para dos claves (datos) diferentes la función de hashing genera el mismo valor. Como no podemos almacenar dos datos diferentes en una misma posición de nuestro vector necesitamos algún mecanismo para resolver las colisiones.

5.1.3 Resolución de Colisiones

En esta sección haremos un repaso muy breve de algunos métodos básicos para resolver colisiones: la búsqueda lineal y el encadenamiento de sinónimos.

Búsqueda lineal (direcciónamiento abierto o hashing cerrado)

La búsqueda lineal es una solución muy sencilla al problema de las colisiones. Cuando queremos almacenar un dato pero su posición está ocupada buscamos linealmente y de forma cíclica hasta encontrar un lugar libre o bien hasta volver al lugar original en cuyo caso la tabla está llena. Cuando queremos recuperar un dato tenemos que tener en cuenta que si en la posición indicada por la función de hashing encontramos otro dato entonces el dato que buscamos podría estar en otra posición por lo que tenemos que buscar linealmente hasta encontrar el dato o bien un lugar vacío. Notemos que si se permiten bajas no podemos detenernos al encontrar una posición dada de baja.

La búsqueda lineal es sencilla pero a medida que aumenta la cantidad de datos en la tabla degradada muy rápidamente y se parece mucho a una simple búsqueda lineal que es $O(n)$ en lugar de $O(1)$.

Encadenamiento de Sinónimos (direcciónamiento cerrado o hashing abierto)

En este método vamos a generar una lista enlazada en cada posición de nuestra tabla de hash para almacenar allí las claves que colisionen. Nuestra estructura es entonces un vector de listas. Cuando queremos almacenar un dato simplemente lo agregamos a la lista que está en la posición indicada por la función de hashing. Cuando queremos buscar un dato accedemos a la lista indicada por la función de hashing y buscamos en la lista.

Este método es muy sencillo y es un poco más eficiente que la búsqueda lineal pero cuando existen muchas colisiones las listas pueden volverse bastante largas por lo que las búsquedas pueden tender a $O(n)$ en lugar de $O(1)$. No obstante es un método muy usado y que debemos tener en cuenta ya que tiene varias aplicaciones interesantes.

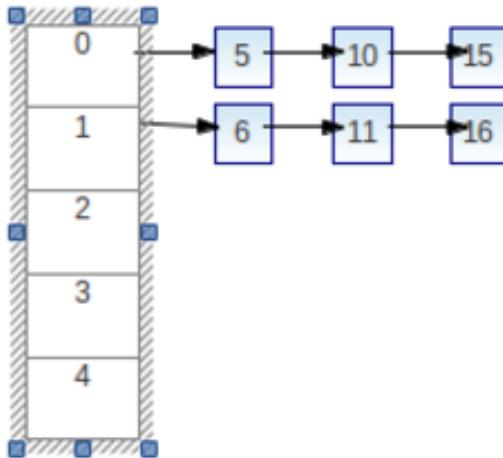


Figure 5.1: Encadenamiento de sinónimos

5.2 Implementación de Diccionarios con Hash Tables

En esta sección describimos el mecanismo usado internamente por Python para el manejo de sus diccionarios. Un diccionario es una estructura de datos en la cual almacenamos datos mediante pares (*clave, valor*). Los diccionarios de Python tienen como requisito que debe poder recuperarse fácilmente un dato a partir de su clave, tienen que poder ser iterables, tienen que soportar altas, bajas y modificaciones y además tienen tamaño dinámico.

Inicialmente al ser creado un diccionario en Python tiene capacidad para 8 (ocho) registros, la posición de los datos en el diccionario está determinada por una función de hashing. Cada tipo de dato tiene asociada una función de hashing, por ejemplo para strings se usa una función como la siguiente:

```

1 arguments: string object
2 returns: hash
3 function string.hash:
4     if hash cached:
5         return it
6     set len to string's length
7     initialize var p pointing to 1st char of string object
8     set x to value pointed by p left shifted by 7 bits
9     while len >= 0:
10        set var x to (1000003 * x) xor value pointed by p
11        increment pointer p
12        set x to x xor length of string object
13    cache x as the hash so we don't need to calculate it again
14    return x as the hash

```

La función devuelve un número de n bits dependiendo de la arquitectura, por ejemplo un número de 64 bits. Dado que inicialmente el diccionario tiene solo 8 posiciones usaremos los últimos 3 bits de la función de hashing para determinar la posición en la cual almacenar el dato.

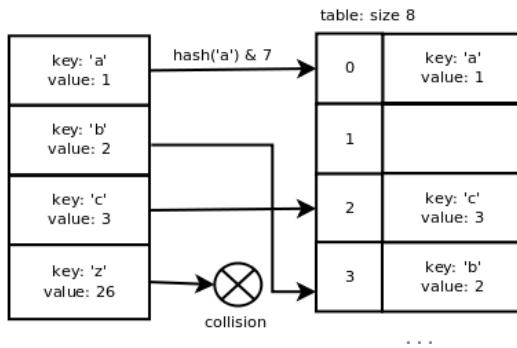


Figure 5.2: Ejemplo de diccionario en Python mostrando una colisión para la posición 3

La función de hashing puede producir colisiones, para resolverlas Python usa un esquema de direccionamiento abierto (hashing cerrado) en el cual las posiciones alternativas a probar son determinadas por una secuencia aleatoria inicializada en base a los primeros bits de la función de hashing notando que estos bits no son los mismos que usamos para determinar la posición. En base a esta secuencia simplemente se "prueban" posiciones hasta encontrar una libre. Este proceso tiene que repetirse cuando se busca un dato en el diccionario, si el dato no está en la posición indicada por la función de hashing hay que probar posiciones alternativas hasta encontrar una posición libre, el dato o bien volver a la posición en la cual comenzamos.

Cuando se da de baja un registro es necesario marcarlo como dado de baja y estos registros "dummy" se consideran libres para dar de alta un nuevo dato pero se consideran ocupados (con basura) al buscar un registro ya que sino podríamos detener la búsqueda en un registro borrado y el dato podría estar mas adelante.

Para minimizar las colisiones la política de los diccionarios de Python es extender el tamaño de la tabla si al menos 2/3 de sus registros están llenos, cuando la cantidad de registros es inferior a 50000 el tamaño de la tabla de se multiplica por 4 y si la cantidad de registros es mayor a 50000 el tamaño se multiplica por 2. Esto quiere decir que si empezamos con 8 registros cuando demos de alta el sexto dato la tabla va a pasar de 8 a 32 registros, cuando tenemos 22 registros el tamaño pasa a 128, con 86 registros el diccionario pasa

a 512 y con 342 registros a 2048. Para hacer esto simplemente se re-ubican las claves en la nueva tabla usando ahora los últimos 5 bits para determinar la posición de los registros. Como la tabla almacena tanto el hash, como la clave y el valor no es necesario volver a hashear cada clave, el valor del hash se encuentra en el diccionario mismo.

Mantener el diccionario disperso, es decir con muchos registros libres es fundamental para reducir el número de colisiones y hacer que las búsquedas y las inserciones sean muy eficientes, el costo es consumir mas memoria de la necesaria pero este costo se reduce ya que a partir de los 50000 registros la capacidad de la tabla se duplica en lugar de cuadruplicar.

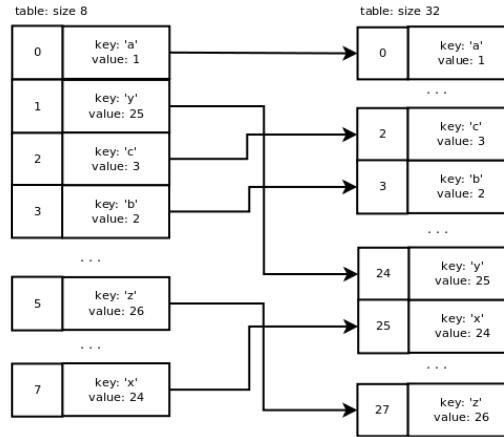


Figure 5.3: Extendiendo un diccionario de 8 a 32 registros

Un diccionario puede contraerse si la cantidad de claves en los mismos es muy inferior al tamaño del mismo. Esto no se hace al dar de baja sino al dar de alta una clave nueva, es decir que la operación insert es la única que puede cambiar el tamaño físico del diccionario.

La forma en que Python maneja sus diccionarios internamente nos permite observar que esquemas de hashing muy simples basados en direccionamiento abierto pueden ser eficientes y permiten el manejo de estructuras dinámicas, es decir que pueden crecer o achicarse a medida que se dan de alta o baja registros. Conocer el funcionamiento interno de los diccionarios es también importante para entender problemas de performance o eficiencia así como también el consumo delta_x VVVVDDXS00000000000009999999999999999999

5.3 Hopscotch Hashing

El método llamado "Hopscotch" tiene como objetivo solucionar algunos de los problemas que observamos en la búsqueda lineal y el encadenamiento de

sinónimos, queremos garantizar $O(1)$ para las búsquedas y queremos que la zona en la cual tenemos que buscar el dato sea igual o muy cercana a la indicada por la función de hashing para favorecer el uso de datos en memoria caché.

En Hopscotch partimos de un vector de "m" elementos que es nuestra tabla de hash, vamos a usar además un número "k" que indica que tan lejos podemos guardar un dato respecto de la posición indicada por la función de hashing. Si $k=3$ podemos guardar el dato en $h(x)$, $h(x)+1$ o $h(x)+2$. Es decir que cada dato tiene k posiciones en las cuales puede almacenarse. Es importante mencionar que esto no quiere decir que si esas k posiciones están ocupadas el dato no pueda almacenarse.

La forma de resolver colisiones en Hopscotch es muy interesante y muy simple. En primer lugar consideraremos las posiciones $h(x)$, $h(x)+1$ hasta $h(x)+k-1$, si alguna de esas posiciones está libre entonces podemos guardar el dato allí sin ningún problema. Si todas las posiciones están ocupadas entonces buscamos linealmente hasta encontrar una posición libre y a partir de esta recorremos hacia atrás viendo si podemos ir moviendo datos hacia adelante de forma de liberar una posición válida para nuestro dato.

Veamos un ejemplo:

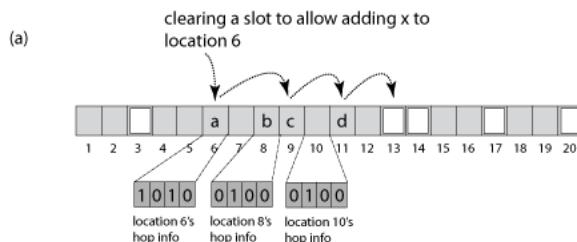


Figure 5.4: Hopscotch Hashing

En nuestro ejemplo $m=21$, $k=4$ y las posiciones en gris están ocupadas. Analizamos el caso de $h(x)=6$. Observamos que las posiciones 6, 7, 8 y 9 están ocupadas por lo que buscamos hacia adelante hasta encontrar la posición libre 13. Buscamos hacia atrás y vemos que $h(d)=11$ por lo que podemos mover "d" a la posición 13 liberando la posición 11, luego vemos que $h(c)=9$ por lo que podemos mover "c" a la posición 11 que ahora está libre. Finalmente podemos almacenar a "x" en la posición 9 que hemos liberado.

Notemos que usando Hopscotch una búsqueda es siempre de tipo $O(k)=O(1)$ ya que el dato, si existe, solo puede estar almacenado en alguna de las k posiciones alternativas para el mismo.

Hopscotch es un método sencillo mucho mas eficiente que la búsqueda lineal o el encadenamiento de colisiones para tablas de hash en memoria.

5.4 Funciones de Hashing

Hemos usado funciones de hashing pero hasta el momento no hemos hablado de como construir una, solo hemos mencionado las funciones de tipo fold and add pero hemos dicho que no son muy eficientes. En esta sección vamos a estudiar algunas funciones de hashing eficientes para strings por lo que empezaremos describiendo cuáles son las propiedades que debe cumplir una función de hashing.

1. La función tiene que ser muy eficiente, es decir generar el resultado en muy poco tiempo
2. La función tiene que producir la menor cantidad de colisiones posibles

El primer punto es muy importante y a menudo pasado por alto, el motivo por el cual usamos una función de hashing es para poder recuperar un dato de forma muy eficiente en tiempo similar a $O(1)$. Si la función de hashing fuera muy compleja e ineficiente podría pasar que buscar linealmente en el vector tarde menos tiempo que usar la función de hashing y acceder a la posición indicada lo cual invalidaría el uso de una función de hashing por completo.

El segundo punto pide minimizar las colisiones, lo cual no solo implica evitar, en lo posible, que nuestras claves colisionen sino que debemos intentar que la distribución de las claves en el espacio de direcciones sea lo mas pareja posible. Es decir que si tenemos 20 datos y nuestro espacio de direcciones es 10 lo deseable es tener 10 colisiones. Si todas las claves fueran a parar a la misma posición tendríamos 19 colisiones que sería la peor función de hashing posible.

En las siguientes secciones explicaremos brevemente algunas funciones de hashing que son muy eficientes y producen pocas colisiones para strings.

5.4.1 FNV

FNV es una función de hashing diseñada por Fowler, Noll y Vo en 1991, se basa en las propiedades de ciertos números primos que denominamos (FNV primes).

Algorithm 15: FNV Hashing

Data: s: string
Result: h: integer

```

1 h = 14695981039346656037;
2 for c in s do
3   h = h * 1099511628211;
4   h = h XOR c;
5 return h;
```

Todas las variables excepto c que es un byte son enteros de 64 bits. En la multiplicación devolvemos los 64 bits inferiores si hay overflow.

5.4.2 Jenkins

La función de Jenkins fue publicada por primera vez en Dr Dobbs en 1997, se basa en realizar operaciones lógicas como XOR y shifts que son muy eficientes en una computadora.

Algorithm 16: Jenkins Hashing

Data: s: string
Result: hash: integer

```

1 for hash=i=0; i<len; i++ do
2   hash += key[i]
3   hash += (hash << 10);
4   hash xor= (hash >> 6);
5   hash += (hash << 3);
6   hash xor= (hash >> 11);
7   hash += (hash << 15);
8 return hash;

```

Por su eficiencia y poca cantidad de colisiones Jenkins es una función muy usada.

5.4.3 Pearson

La función de Pearson no es tan usada como Jenkins o FNV pero tiene varias propiedades interesantes que hacen que su estudio sea conveniente. En primer lugar vamos a explicar la versión de Pearson para generar un hash de 8 bits, luego veremos como podemos extender esto.

Para la función vamos a necesitar una tabla que contenga una permutación aleatoria de los números 0 a 255 es decir una tabla de 256 posiciones.

Algorithm 17: Pearson Hashing

Data: s: string
Result: hash: integer

```

1 h = 0;
2 for c in s do
3   index = h xor c;
4   h = T[index];
5 return h

```

Hay que destacar que cualquier cambio en cualquier bit de la clave produce un resultado diferente, por esto es sencillo extender la función si necesitamos un hash de mas de 8 bits, simplemente incrementamos el primer byte del string en 1 y aplicamos nuevamente la función, de esta forma obtenemos dos hashes

de 8 bits que podemos concatenar para un único hash de 16 bits. De la misma forma podemos generar hashes de 24,32,64,196 bits,etc.

La función de Pearson nos permite también una primera aproximación rudimentaria a una función de hashing perfecta, si tenemos pocas claves a hashear podemos simplemente probar diferentes tablas al azar hasta que alguna de las tablas nos genere una función de Pearson que sea perfecta. Este es un algoritmo aleatorizado de tipo Las Vegas ya que siempre produce un resultado exacto pero con una cierta probabilidad puede tardar mucho tiempo en generar el resultado.

5.4.4 Funciones criptográficas

Las funciones de hashing criptográficas son funciones de hashing que deben cumplir requisitos adicionales que hacen a la seguridad de las mismas. Una función de hashing criptográfica debe cumplir con las siguientes características:

1. La función tiene que producir la menor cantidad de colisiones posibles
2. Dado $h(x)$ tiene que ser muy difícil hallar x
3. Tiene que ser muy difícil hallar x e y de forma tal que $h(x)=h(y)$
4. Un cambio mínimo en la clave tiene que producir un cambio significativo en el resultado (efecto avalancha)

La construcción de funciones de hashing criptográficas es un asunto muy delicado, hoy en día la función de hashing criptográfica mas usada es SHA-256 por lo que vamos a explicar el funcionamiento de la misma.

SHA-256 se basa en la combinación de dos primitivas criptográficas muy importantes: La construcción de Merkle-Damgard y la construcción de Davis-Meyer.

Construcción de Merkle-Damgard

En esta construcción suponemos que existe una función " f " que recibe un bloque de m bits y devuelve un bloque de n bits con $n < m$. A esta función se la llama función de compresión aunque no tiene nada que ver con compresión de datos. Simplemente devuelve menos bits que los que recibe.

El objetivo de la construcción de Merkle-Damgard es dado un string de una cantidad arbitraria de bits emitir un único resultado de tamaño igual al tamaño que emite la función f .

Supongamos que " n " es la cantidad de bits que emite nuestra función de compresión f , dividimos nuestro mensaje entonces en bloques de " n " bits agregando un padding al final en caso de ser necesario. El padding es crítico para el correcto funcionamiento de la función y no debe ser fijo.

Como la función de compresión recibe bloques de m bits agregamos un bloque ficticio llamado IV con el cual podemos comenzar el algoritmo. Notar que la construcción es muy simple y se basa en concatenar el resultado anterior de f al bloque próximo y luego volver a aplicar f .

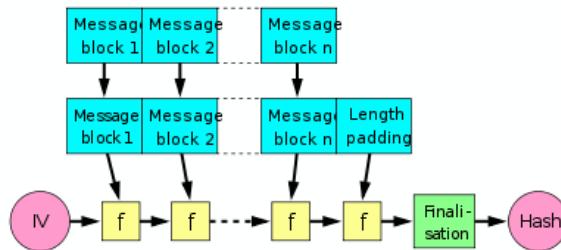


Figure 5.5: Construcción de Merkle-Damgård

Mediante la construcción de Merkle-Damgård podemos aplicar una función de compresión cualquiera a un mensaje de longitud arbitraria de forma segura, veamos ahora como construir una función de compresión segura.

Construcción de Davis-Meyer

La construcción de Davis-Meyer permite generar una función de compresión segura a partir de un algoritmo de encripción seguro. El procedimiento es muy simple, recibimos dos bloques y usaremos uno como clave para encriptar el otro finalizando con el xor entre el bloque 1 y el resultado de la encripción.

SHA-256 es entonces una construcción de Merkle-Damgård en donde la función de compresión es una construcción de Davis-Meyer usando SHACAL-2 como algoritmo de encripción. Cada bloque del mensaje tiene 512 bits y la función de compresión devuelve 256 bits.

Es importante destacar que es un error muy grave usar una función de hashing tradicional cuando se necesita una función de hashing criptográfica pero también es un error grave usar una función criptográfica cuando solo necesitamos una función de hashing tradicional, esto es porque las funciones criptográficas son menos eficientes y estaríamos perdiendo velocidad a cambio de propiedades adicionales que no necesitamos!

5.5 Hashing Universal

En muchos algoritmos necesitamos no una sino varias funciones de hashing, con lo que hemos visto esto no es una tarea sencilla ya que podríamos usar Jenkins, FNV y alguna otra pero tarde o temprano se nos terminarían las funciones. El concepto de Hashing Universal pasa por construir no una función de hashing sino una familia de funciones de hashing de forma tal que podamos elegir las funciones que necesitemos entre el total de funciones que constituyen dicha familia.

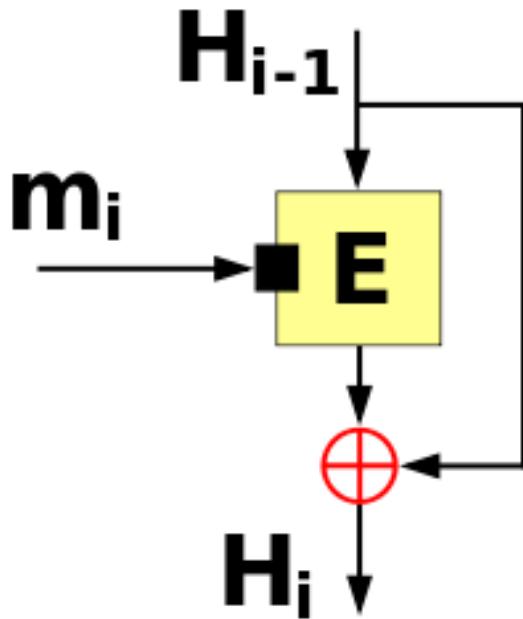


Figure 5.6: Construcción de Davis-Meyer

5.5.1 Definición de Hashing Universal

Sea \mathcal{H} una familia de funciones de hashing de la forma $h \in \mathcal{H} : U \rightarrow [0..m - 1]$ es decir que recibimos una clave cualquiera dentro de U y generamos un número entre 0 y $m - 1$. Decimos que \mathcal{H} es Universal si:

$$\forall x, y \in U, h \in \mathcal{H}, x \neq y$$

$$P[h(x) = h(y)] \leq \frac{1}{m}$$

Es decir que para cualquier función de hashing de nuestra familia la probabilidad de que dos claves diferentes colisionen debe ser menor o igual a $1/m$ siendo m el espacio de direcciones de cada una de las funciones de hashing de la familia.

5.5.2 Hashing Universal para valores atómicos (numéricos)

Vamos a ver dos versiones que nos permiten crear una familia de funciones de hashing universal para valores numéricos. La primer versión es la mas clásica y se denomina función de Carter-Wegman:

$$h \in \mathcal{H} \Rightarrow h(x) = (a * x + b \pmod{p}) \pmod{m}$$

En donde p es un número primo mayor o igual a m , a es un número entre 1 y $p - 1$ y b es un número entre 0 y $p - 1$

La segunda construcción está basada en la función FNV:

```

2 def hash( d, str ):
3     if d == 0: d = 0x01000193
4
5     for c in str:
6         d = ( (d * 0x01000193) ^ ord(c) ) & 0xffffffff;
7
8     return d

```

Como podemos ver en la segunda construcción parametrizamos la función en base a un número d .

5.5.3 Hashing Universal para claves de longitud fija

Vamos a plantear ahora una familia de funciones de hashing que funcione para datos de longitud fija, pueden ser números, strings de longitud fija o vectores. Sea m , nuestro espacio de direcciones, p un número primo mayor o igual a m y sean claves x compuestas por $r + 1$ números: $x_0..x_r$. Vamos a elegir $r + 1$ números a_i entre 1 y $m - 1$ y nuestra función de hashing quedará definida por:

$$h \in \mathcal{H} \Rightarrow h(x) = \sum_{i=0}^r (a_i * x_i \pmod p) \pmod m$$

A efectos de simplificar el análisis supongamos que m es un número primo en cuyo caso podemos usar:

$$h \in \mathcal{H} \Rightarrow h(x) = \sum_{i=0}^r a_i * x_i \pmod m$$

Notemos que hemos definido una familia de funciones de hashing ya que por cada selección de los parámetros a_i creamos una función diferente. En concreto como cada a_i puede tomar un valor entre 0 y $m - 1$ hay un total de m^{r+1} funciones de hashing en nuestra familia. Vamos a intentar demostrar ahora que H es universal.

Para demostrar esto vamos a partir de dos claves diferentes x e y cada clave está formada por $r + 1$ elementos, sin perder generalidad podemos pedir que x e y difieran en su primer elemento es decir $x_0 \neq y_0$ y esto es suficiente para garantizar que las claves sean diferentes. Recordemos que m es un número primo y que tenemos que elegir $r + 1$ números a_i entre 0 y $m - 1$. Entonces planteamos:

$$h(x) = h(y) \pmod m$$

$$\sum_{i=0}^r a_i * x_i = \sum_{i=0}^r a_i * y_i \pmod m$$

$$\begin{aligned}
(a_0 * x_0) + \sum_{i=1}^r a_i * x_i &= (a_0 * y_0) + \sum_{i=1}^r a_i * y_i \pmod{m} \\
(a_0 * x_0) + \sum_{i=1}^r a_i * x_i - \sum_{i=1}^r a_i * y_i &= 0 \pmod{m} \\
a_0(x_0 - y_0) + \sum_{i=1}^r a_i * x_i - \sum_{i=1}^r a_i * y_i &= 0 \pmod{m} \\
a_0(x_0 - y_0) + \sum_{i=1}^r a_i * (x_i - y_i) &= 0 \pmod{m} \\
a_0(x_0 - y_0) &= -\sum_{i=1}^r a_i * (x_i - y_i) \pmod{m}
\end{aligned}$$

Una vez que hemos llegado hasta aquí tenemos que demostrar que solo existe un valor posible para a_0 entre 0 y $m-1$ que cumple con nuestra última ecuación. El lado derecho de la ecuación es algún número y sabemos que $x_0 - y_0 \neq 0$ por precondición. Por lo tanto tenemos algo de la forma:

$$a_0 * z = y \pmod{m}$$

Supongamos que existen dos valores posibles para a_0 que cumplan con esto:

$$\begin{aligned}
a_{01} * z &= y \pmod{m} \\
a_{02} * z &= y \pmod{m}
\end{aligned}$$

Restando:

$$(a_{01} - a_{02}) * z = 0 \pmod{m}$$

Y siendo m un número primo esto es absurdo porque a_{01} y a_{02} son distintos y z no puede ser cero. (z era $x_0 - y_0$).

Esto demuestra que la familia de funciones propuesta efectivamente define una familia universal.

5.5.4 Hashing Universal para claves de longitud variable

Consideramos ahora el caso en el cual nuestras claves tienen longitud variable y no hay una longitud máxima definida. Si existiera una longitud máxima para las claves entonces podríamos simplemente hacer padding y usar la familia de funciones anterior sin perder la universalidad de la misma.

Planteamos la siguiente familia de funciones de hashing para una clave de longitud l

$$h \in \mathcal{H} \Rightarrow h(x) = h_{int}\left(\left(\sum_{i=1}^l x_i * a^i\right) \bmod p\right)$$

Donde p es un número primo grande, a es un número aleatorio entre 0 y $p - 1$ y h_{int} es una función de hashing tomada de la familia universal que definimos anteriormente que recibe un número entre 0 y $p - 1$ y devuelve otro número entre 0 y $m - 1$ siendo m nuestro espacio de direcciones.

Notemos que podemos calcular rápidamente una función cualquiera de esta familia sin necesidad de elevar a a ninguna potencia:

$$\sum_{i=1}^l x_i * a^i = x_0 * a^0 + x_1 * a^1 + x_2 * a^2 + \dots + x_n * a^n$$

$$x_0 + a(x_1 + a * x_2 + a^2 * x_3 + \dots + a^{n-1} * x_{n-1})$$

De aquí podemos derivar una fórmula iterativa para calcular el resultado:

Algorithm 18: Hashing Universal para Strings de Longitud Variable

Data: s: string
Result: h: integer

```

1 h = x0;
2 for i=0; i<l; ++i) do
3   h = ((h * a) + xi) mod p
4 return hint(h)

```

Tenemos que probar que nuestra familia es universal. Consideremos dos strings diferentes: x e y y paddeamos la longitud del mas corto a la del mas largo: l . Para que exista una colisión antes de llamar a h_{int} debería ocurrir que:

$$\sum_{i=1}^l x_i * a^i = \sum_{i=1}^l y_i * a^i \pmod{p}$$

$$\sum_{i=1}^l a^i(x_i - y_i) = 0 \pmod{p}$$

Es decir que a tiene que ser raiz del polinomio con coeficientes $x_i - y_i$ el polinomio tiene a lo sumo l raices modulo p por lo tanto la probabilidad de colisión es a lo sumo l/p .

Como sabemos que la probabilidad de colisión de h_{int} es $1/m$ la probabilidad total de colisiones en nuestra familia es:

$$\frac{1}{m} + \frac{l}{p}$$

Y cuando p es un número lo suficientemente grande esto es igual a $1/m$ por lo tanto la condición para que esta familia de funciones de hashing sea universal es que el número primo p sea significativamente mas grande que la longitud máxima de las claves que vamos a procesar.

5.6 Cuckoo Hashing

Vamos a explicar nuestro primer algoritmo basado en múltiples funciones de hashing. El algoritmo llamado "Cuckoo Hashing" fue creado por Pagh y Rodler en el año 2001 [Pagh-Rodler]. Se basa en el uso de mas de una función de hashing, para empezar vamos a usar dos.

Cada función de hashing va a definir una posición alternativa para nuestro dato en una tabla de hash. El algoritmo de búsqueda es muy simple, el dato tiene que estar en alguna de las dos posiciones alternativas o no está en la tabla es decir que es $O(1)$. La inserción funciona de forma muy sencilla: almacenamos el dato en la posición indicada por la primer función de hashing, si ese lugar estaba ocupado entonces reubicamos ese dato en su posición alternativa y así sucesivamente hasta que no hay mas datos a reubicar o bien entramos en un loop.

Ejemplo: Definamos una tabla de 6 posiciones (0 a 5) y daremos el valor de dos funciones de hashing $h1$ y $h2$ para cada clave a insertar.

Tabla inicial: $_ , _ , _ , _ , _ , _$

$h1(A)=2, h2(A)=0$

$_ , _ , A , _ , _ , _$

$h1(B)=4, h2(B)=1$

$_ , _ , A , _ , B , _$

$h1(C) = 2, h2(C) = 5$

$_ , _ , C , _ , B , _ \quad (A)$

$A , _ , C , _ , B , _$

$h1(D)=2, h2(D)=3$

$A , _ , D , _ , B , _ \quad (C)$

$A , _ , D , _ , B , C$

$h1(E) = 5, h2(E)=0$

$A , _ , D , _ , B , E \quad (C)$

$A , _ , C , _ , B , E \quad (D)$

$A , _ , C , D , B , E$

$h1(F)=3, h2(F)=4$

$A , _ , C , F , B , E \quad (D)$

$A , _ , D , F , B , E \quad (C)$

$A , _ , D , F , B , C \quad (E)$

$E , _ , D , F , B , C \quad (A)$

$E , _ , A , F , B , C \quad (D)$

$E , _ , A , D , B , C \quad (F)$

$E , _ , A , D , F , C \quad (B)$

E,B,A,D,F,C

Es importante estudiar algún mecanismo para detectar si es posible o no dar de alta un nuevo dato sin necesidad de detectar que nuestro algoritmo ha entrado en loop. Para hacer esto es posible construir un grafo en el cual hay tantos nodos como posiciones en nuestra tabla de hash y por cada dato que insertamos construimos una arista que une las dos posiciones candidatas indicadas por la función de hashing.

Por ejemplo si nuestra tabla de posiciones para nuestros datos fuera la siguiente:

h1	h2
0	6
1	6
2	0
3	1
5	3
4	4
6	3
7	4
8	0

Table 5.1: Ejemplo Cuckoo Hashing.

El grafo construido tendría la siguiente forma.

Para que pueda darse de alta el grafo resultante luego de la operación debe ser un pseudo-bosque. Un pseudo-bosque es un conjunto de pseudo-árboles y un pseudo-árbol es un grafo que puede tener un ciclo pero no dos. En resumen en nuestro grafo cada componente conexo admite un ciclo pero no dos. En el caso de nuestro ejemplo podemos dar de alta datos que tengan claves de tipo (7,8) (7,5) (4,2) etc pero no podríamos dar de alta claves con hashes (0,3) (1,2) (5,6) (4,7) etc porque se formaría un segundo ciclo en alguno de los componentes conexos.

El método del Cuckoo puede hacerse mas eficiente admitiendo mas de un registro por bucket (posición en la tabla de hash) o bien usando 3 funciones de hashing en lugar de 2. Con buckets para 2 registros y 2 funciones de hashing es posible llenar mas de un 90% de la tabla con muy baja probabilidad de que una inserción falle.

La nomenclatura a usar es $C_{n,m}$ en donde n es la cantidad de funciones de hashing a usar y m es la cantidad de registros por bucket. Por ejemplo en $C_{2,8}$ usamos dos funciones de hashing con buckets que soportan 8 claves cada uno. Varios esquemas de Cuckoo Hashing logran eficiencia cercana al 95% del espacio por lo que podemos generar tablas de hash bastante compactas.

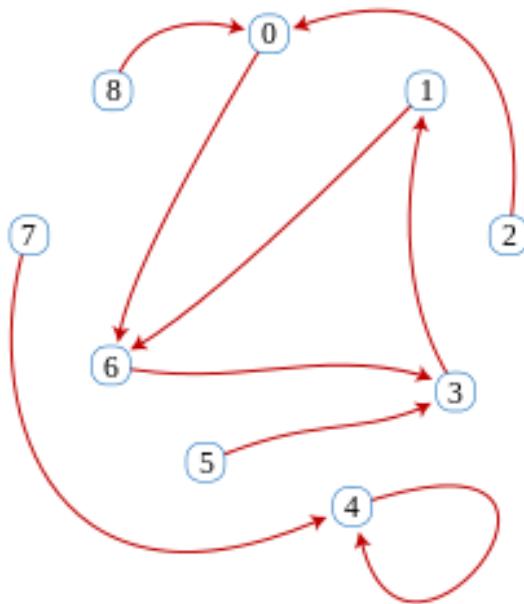


Figure 5.7: Cuckoo Hashing

	b=1	b=2	b=3	b=4
4 hash	97%	99%	99.9%	99.9%
3 hash	91%	97%	98%	99.9%
2 hash	49%	86%	93%	96%

Table 5.2: Capacidad de carga de una tabla de Cuckoo hashing según la cantidad de funciones de hashing a usar y el tamaño de los buckets.[Pagh-Rodler]

5.6.1 Cuckoo Hashing With a Stash

Cuando una inserción falla una alternativa muy simple es poner el registro en cuestión en un área de overflow o *stash*, esto permite que la clave se almacene y que el método continue siendo eficiente. Con áreas de stash relativamente pequeñas el método del Cuckoo puede lograr una eficiencia muy importante. El costo, obvio, es tener que buscar las claves en el Stash en caso de no encontrarlas en las posiciones que indique la función de hashing, curiosamente si el tamaño del stash es constante esto no afecta el orden del algoritmo ya que $O(2) = O(2 + k)$ siendo k el tamaño del stash una constante.

5.6.2 Estructuras de Datos basadas en Cuckoo Hashing

Las tablas de hash generadas por el método del Cuckoo tienen una propiedad muy interesante: una misma clave solo puede estar en alguna de sus posiciones alternativas en cualquier tabla de hash generada por el método. Esto sirve cuando tenemos que representar datos en varias tablas de hash y luego queremos hacer alguna operación entre las tablas.

Consideremos el caso de almacenar vectores dispersos en un diccionario, un vector disperso como por ejemplo $[0,0,1,0,3,0,0,-1,0,0,0,4,0,-2]$ puede representarse mediante tuplas de tipo (clave,valor) en donde la clave es la dimensión. El vector de nuestro ejemplo quedaría representado por 5 (cinco) tuplas: $(3,1);(5,3);(8,-1);(12,4)$ y $(14,-2)$ numerando las dimensiones a partir de 1. Podemos almacenar un vector disperso en un diccionario usando Cuckoo hashing, por ejemplo supongamos que usamos tablas para 3 claves con buckets de tamaño 2 y dos funciones de hashing lo cual nos da una capacidad máxima para 6 claves.

Supongamos los siguientes valores para las dos funciones de hashing:

key	h1	h2
3	1	0
5	2	1
8	0	2
12	0	1

Table 5.3: Funciones de hashing para el ejemplo

Luego de realizar las inserciones usando Cuckoo la tabla nos quedaría:

bucket		
0	$(14,-2)$	$(12,4)$
1	$(3,1)$	
2	$(5,3)$	$(8,-1)$

Table 5.4: Vector disperso como Cuckoo table

Supongamos ahora que tenemos otro vector almacenado en una tabla similar. Es posible realizar ciertas operaciones algebraicas entre los vectores de forma eficiente gracias a las propiedades del Cuckoo [Zhou-Li-Andersen-Smola]. Por ejemplo consideremos el producto interno entre dos vectores representados mediante el Cuckoo.

Podemos recorrer la primera tabla de forma secuencial, tomamos el primer bucket y encontramos $(14,-2)$ ahora si el segundo vector tiene la clave 14 entonces estará en el primer bucket o bien en su posición alternativa; buscamos y si encontramos 14 en el segundo vector entonces multiplicamos los valores y vamos acumulando. Este proceso lo repetimos, en total la operación es $O(n)$ ya que por cada bucket de la tabla solo usamos una función de hashing para verificar su posición alternativa en la segunda tabla. El producto interno es la

operación mas usada por ejemplo en Kernels por lo que para vectores dispersos de grandes dimensiones este método es muy interesante ya que como hemos visto la representación del Cuckoo es muy eficiente.

Otras operaciones como por ejemplo la suma son mas complejas ya que la cantidad de claves resultantes puede ser mayor a la cantidad que teníamos en cada tabla y podrían no entrar en una nueva tabla.

5.7 Balls & Bins

Es posible generalizar el método del cuckoo a un esquema genérico en el cual usemos múltiples funciones de hashing. Consideremos primero el caso mas simple en el cual tenemos una tabla de hash en donde cada bucket tiene capacidad para k datos. Usando dos funciones de hashing tenemos dos buckets candidatos para cada clave y podemos tomar como heurística almacenar el dato en el bucket que tenga menor cantidad de datos. Este esquema se denomina Two-Way hashing. Es un esquema bastante eficiente y que requiere muy pocos detalles de programación.

Si se quisiera buscar aun mas eficiencia se podría desarrollar un grafo similar al del Cuckoo en donde indicamos por cada dato cuáles son sus posiciones alternativas con una arista. Con este grafo es posible realizar un balanceo de los buckets de forma tal que tengan todos una cantidad pareja de datos. Esto se logra simplemente realizando movimientos en el grafo que consisten en invertir la dirección de las aristas.

Por ejemplo consideremos el caso en el cual queremos insertar un dato cuyas posiciones alternativas son 3 y 5 y tenemos 2 datos en cada una de estas posiciones, si existe alguna otra posición con 1 solo dato es posible que exista una secuencia de movimientos que permita liberar la posición 3 o 5 de alguno de sus datos, es fácil determinar esta cadena de movimientos observando el grafo formado.

Por ejemplo supongamos que luego de una inserción obtenemos el siguiente grafo (fig 4.6). Vemos que uno de los buckets tiene 3 registros pero hay otros con solo 1 o ningún registro.



Figure 5.8: Balls & Bins Antes

Invertiendo la dirección de la arista UV logramos que ninguno de los buckets tenga mas de 2 registros, esta operación consiste simplemente en mover un dato a su posición alternativa. En otros casos hace falta realizar mas de un movimiento para lograr el balanceo.

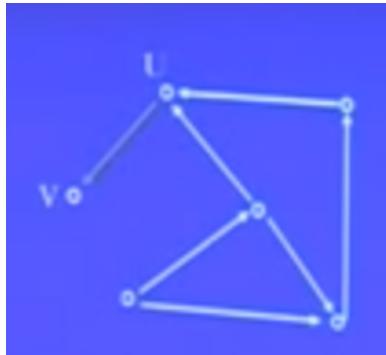


Figure 5.9: Balls & Bins Después

El esquema de balls & bins usando balanceo logra una eficiencia superior al 95% pero tiene como contra la necesidad de implementar el algoritmo de balanceo. En muchos casos el esquema mas simple de two-way hashing es preferible ya que es mucho mas simple de implementar y logra buenos niveles de eficiencia.

5.8 Hashing Perfecto

Una función de hashing perfecta es aquella que no tiene colisiones, esto garantiza $O(1)$ para cualquier búsqueda de claves. En esta sección veremos dos mecanismos diferentes para crear una función de hashing perfecta.

5.8.1 Esquema FKS

Empecemos pensando una forma muy ineficiente de realizar hashing perfecto: usando una tabla de hashing muy grande. Por ejemplo si tenemos m claves a insertar queremos saber cuál debería ser el tamaño de una tabla de hash de forma tal que la probabilidad de una colisión sea menor a $1/2$, haciendo algunas cuentas llegamos a que la tabla debe tener un espacio de direcciones de m^2 . Si llamamos m al espacio de claves y n al espacio de direcciones entonces la probabilidad de una colisión es m/n , si $n = m^2$ entonces la probabilidad de una colisión es $1/m$ que es lo que queremos.

El problema es que si tenemos 1 millón de claves no podemos tener una tabla de hash con capacidad para 1 billón de datos porque es un desperdicio muy grande de espacio y probablemente no tengamos lugar suficiente.

La solución a este problema es usar dos funciones de hashing en lo que conocemos como esquema FKS [Fredman-Komlos-Szemeredi], nuestra primera

función de hashing va a tener un espacio de direcciones que podemos fijar como parámetro, llamémoslo k siendo k un número primo cercano a m a efectos de simplificar suponemos $k = m$.

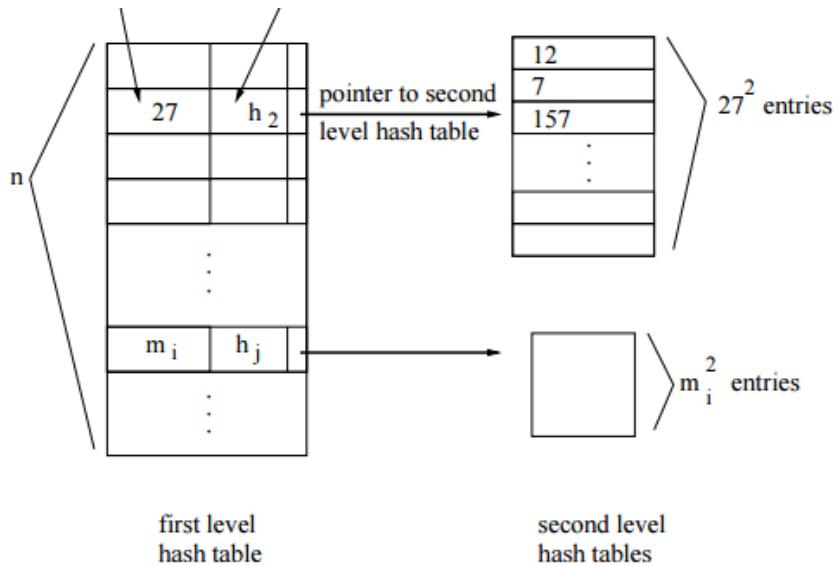


Figure 5.10: Esquema FKS

Cada uno de los $k = m$ buckets apuntados por la primera función de hashing contiene un número que es la cantidad de claves que hasheadaron a esa posición (m_i), y una *segunda* función de hashing cuyo espacio de direcciones es m_i^2 (o el número primo más cercano a m_i^2). Esta función de hashing la tomamos de una familia universal hasta estar seguros de que no tenemos colisiones.

Por ejemplo supongamos que tenemos las claves: 1,2,7,8,9,11,12,13,16,17,46 (11 claves) y usamos como función de hashing de primer nivel $2x \bmod 11$. La siguiente tabla nos muestra la distribución de las claves, y la segunda función de hashing asociada a cada bucket.

Supongamos que buscamos la clave "12", aplicamos la primer función de hashing $2x \bmod 11$ y nos da $2 * 12 \bmod 5 = 2$ accedemos al bucket 2 de la tabla de primer nivel y recuperamos la segunda función de hashing: $x \bmod 13$ junto con un puntero a la tabla de la segunda función, por lo tanto buscamos el 12 en la posición $12 \bmod 13 = 12$ de la segunda tabla y lo encontramos en $O(2) = O(1)$.

El esquema FKS garantiza $O(1)$ para la recuperación de claves y el costo de espacio es $2m$. El costo de la primera tabla es m y el de la segunda tabla es $\sum_{i=0}^{m-1} m_i^2$ el número de colisiones en la primera tabla puede calcularse contando los pares de elementos en la segunda tabla entonces nos queda:

K	Keys	h2	table
0	11	x	[11]
1	17	x	[17]
2	1,12,46	$x \bmod 13$	[-, 1, -, -, -, -, -, 46, -, -, -, -, 12]
3	7	x	[7]
4	2,13	$x \bmod 5$	[-, -, 2, 13, -]
5	8	x	[8]
6			
7	7	x	[7]
8			
9			
10	16	x	[16]

Table 5.5: Esquema FKS

$$m + 2 \sum_{i=0}^{m-1} \binom{m_i}{2} = n + \frac{2 \binom{m}{2}}{m} < 2m \quad (5.1)$$

5.8.2 Hashing Perfecto Dinámico

Si podemos dar altas y bajas de claves entonces el esquema anterior falla. Podemos usar un mecanismo muy simple basado en agrandar y achicar el tamaño de una función perfecta.[Dietzfelbinger]

Si tenemos m claves construimos una función perfecta para $2m$ claves usando el esquema FKS, esta sirve hasta que tengamos $2m$ claves en cuyo caso reconstruimos la función de hashing para $4m$ elementos.

Como antes si tenemos una colisión en la tabla de segundo nivel simplemente reconstruimos la tabla eligiendo otra función de hashing de nuestra familia \mathcal{H} , esto no ocurre muy seguido por lo que el costo es bajo.

Si eliminamos elementos hasta que tenemos $m/4$ claves en total entonces reconstruimos la tabla para m (no para $2m$) ya que si borramos lo suficiente podemos reconstruir a la mitad del tamaño.

La posibilidad de mantener una función de hashing perfecta con bajo costo para datos dinámicos es una de las características más importantes del esquema FKS.

5.8.3 Hashing Perfecto y Mínimo: Hash and Displace

En estas secciones vamos a desarrollar la construcción de una función de hashing perfecta y mínima es decir que tengamos $O(1)$ para las consultas y $O(m)$ espacio. Esto sólo es posible cuando conocemos cuáles son todas las claves que queremos almacenar en la tabla y los datos son estáticos. Cuando eso pasa es posible desarrollar una función que mapee cada clave a una única posición de forma tal de no tener colisiones.

Vamos a ver en primer lugar un algoritmo conocido como HDC (Hash Displace Compress) cuyo origen podemos rastrear a [Pagh], [Botelho-Pagh-Ziviani] y [Bellazzougui-Botelho-Dietzfelbinger]. Este algoritmo está fuertemente inspirado en el esquema FKS y es altamente eficiente para generar funciones de hashing perfectas y mínimas.

El funcionamiento del algoritmo es bastante sencillo, necesitamos una familia de funciones de hashing universales que podamos parametrizar con un número entero $h(i, x)$, las funciones de esta familia nos generan un número entre 0 y $n - 1$ que es la cantidad de claves que tenemos.

En un primer paso vamos a hashear todas las claves a una tabla G usando $h(0, x)$ encadenando los sinónimos. El resultado es una simple tabla de hash con encadenamiento de sinónimos.

Vamos a usar un vector de n bits para marcar qué valores ya hemos usado en la construcción de la función de hashing perfecta, inicialmente este vector está en 0.

En un segundo paso vamos a recorrer esta tabla ordenando los buckets que tengan 2 o mas claves de acuerdo a la cantidad de sinónimos en los mismos de mayor a menor, es decir que procesamos primero el bucket (lista) con mayor cantidad de colisiones.

Por cada bucket probamos funciones de hashing $h(i, x)$ con $i = 1, 2, \dots, etc$ hasta encontrar una función que nos distribuya las claves en posiciones que aun no hemos usado para nuestra función de hashing. Notemos que inicialmente no hemos usado ninguna función por lo que es muy probable que encontremos rápidamente una función de hashing que no produzca colisiones para las claves del primer bucket, frecuentemente $h(1, x)$ funciona. Luego seguimos con el resto de los buckets teniendo en cuenta que las posiciones que ya hemos usado anteriormente no podemos volver a usarlas. En cada caso una vez que encontramos la función en la tabla G ponemos el valor de i que corresponde a la misma.

Luego del segundo paso solo nos quedan los buckets que tienen un solo registro, estos los vamos a distribuir en las posiciones que quedaron libres simplemente indicando la posición en la tabla G , para distinguir este caso de algún valor de i lo vamos a marcar con un número negativo.

La función $h(0, x)$ y la tabla G nos determinan entonces la función de hashing perfecta y mínima.

Para buscar una clave usando esta función usamos primero $h(0, x)$, esto nos da una posición en la tabla G , accediendo a esta tabla observamos si el número allí almacenado es positivo o negativo, si es negativo entonces el valor absoluto del mismo nos da el resultado de la función de hashing. Si el número es en cambio positivo z el valor de la función de hashing es $h(z, x)$.

Consideremos un ejemplo muy simple para verificar el funcionamiento del algoritmo.

Comenzamos con 10 claves que, de forma imaginativa, llamaremos a,b,c,d,e,f,g,h,i,j.

Supongamos que la función de hashing $h(0, x)$ nos da los siguientes resultados:

$$h(0, a) = 4$$

```

h(0,b) = 2
h(0,c) = 3
h(0,d) = 1
h(0,e) = 2
h(0,f) = 5
h(0,g) = 3
h(0,h) = 2
h(0,i) = 5
h(0,j) = 0

```

Tenemos entonces que el bucket 2 tiene 3 claves (b,e,h), el bucket 3 tiene dos claves (c,g), el bucket 5 tiene 2 claves (f,i) y luego tenemos tres buckets con una clave cada uno 4:(a), 1:(d) 0:(j).

Empezamos la segunda fase con el bucket 2 entonces que tiene 3 claves, probamos $h(1,x)$

```

h(1,b) = 4
h(1,e) = 7
h(1,h) = 9

```

Como hasta ahora no hemos usado ninguno de los 10 valores disponibles para la función de hashing perfecta el valor $i = 1$ funciona perfectamente para este bucket. Entonces nos queda $G(2) = 1$. Y marcamos las posiciones 4,7 y 9 como usadas para la función perfecta.

Pasamos ahora al bucket 3 con 2 claves (c,g) probamos la función con $i = 1$

```

h(1,c) = 5
h(1,g) = 4

```

Lamentablemente el valor 4 ya fue usado anteriormente por lo que tenemos que rechazar esta función y probar la siguiente. Estamos buscando 3 valores diferentes entre sí y distintos de 4,7 o 9.

```

h(2,c) = 0
h(2,g) = 3

```

Estos valores no estaban usados así que aceptamos esta función, marcamos $G(3) = 2$ y marcamos como usandos los valores 0 y 3.

El siguiente bucket es el 5 (f,i) y los valores que no podemos aceptar son 4,7,9,0,3

```

h(1,f) = 9
h(1,i) = 2

```

```

h(2,f) = 8
h(2,i) = 8

```

```

h(3,f) = 3
h(3,i) = 6

```

$$\begin{aligned} h(4, f) &= 6 \\ h(4, i) &= 1 \end{aligned}$$

Nos queda entonces $G(5) = 4$ y hemos usado las posiciones 4,7,9,0,3,6,1. Nos quedan solo los buckets de una clave que vamos a poner en las posiciones que nos quedaron libres. Los buckets son: 4(a), 1(d) y 0(j). Y nos quedaron libres las posiciones 2,5 y 8. Entonces podemos hacer $G(4) = -2$, $G(1) = -5$ y $G(0) = -8$

Verifiquemos ahora que la función generada es perfecta:

$$\begin{aligned} h(0, a) &= 4, G(4)=-2 \Rightarrow 2 \\ h(0, b) &= 2, G(2)=1, h(1, b)=4 \Rightarrow 4 \\ h(0, c) &= 3, G(3)=2, h(2, c)=0 \Rightarrow 0 \\ h(0, d) &= 1, G(1)=-5 \Rightarrow 5 \\ h(0, e) &= 2, G(2)=1, h(1, e)=7 \Rightarrow 7 \\ h(0, f) &= 5, G(5)=4, h(4, f)=6 \Rightarrow 6 \\ h(0, g) &= 3, G(3)=2, h(2, g)=3 \Rightarrow 3 \\ h(0, h) &= 2, G(2)=1, h(1, h)=9 \Rightarrow 9 \\ h(0, i) &= 5, G(5)=4, h(4, i)=1 \Rightarrow 1 \\ h(0, j) &= 0, G(0)=-8 \Rightarrow 8 \end{aligned}$$

Como podemos ver hemos asignado cada clave a una posición única de 0 a 9 por lo que la función resultante es perfecta y mínima. Observemos que la función sigue siendo O(1) ya que solo usamos una o dos funciones de hashing para obtener el dato.

El análisis de este algoritmo nos muestra que el paso crítico es procesar los buckets ordenados de mayor a menor según la cantidad de sinónimos, esto reduce notablemente la cantidad de funciones de hashing a probar hasta encontrar alguna que no produzca colisiones para dicho bucket. Este algoritmo puede complementarse comprimiendo los números en la tabla G para que ocupen aun menos espacio, esto es factible ya que en general se trata de números pequeños en el caso de los positivos y de números cuyo valor absoluto es ascendente para los negativos, aprovechando estas características podemos usar alguna técnica de compresión de datos para que los números ocupen poco lugar minimizando la cantidad de memoria necesaria para la función de hashing.

5.8.4 Hashing Perfecto y Mínimo: Algoritmo II

Si tenemos un total de n claves el primer paso es elegir un número m mayor a n y aplicarle a cada clave dos funciones de hashing que conviertan las claves al espacio de direcciones $0..m-1$. Esto lo hacemos, por supuesto, generando dos funciones de hashing de una familia universal.

ejemplo, tenemos 11 claves y elegimos $m=16$.

El siguiente paso es armar un grafo de m nodos en donde cada clave es una arista y los valores de las funciones de hashing determinan los nodos que unimos con cada arista. Rotulamos cada arista con el número de clave correspondiente: la arista 0 es la clave "A", la arista 1 la clave "B", etc...

clave	h1	h2
A	2	9
B	6	1
C	4	12
E	14	11
H	3	11
U	7	6
J	10	15
O	6	11
R	12	14
S	5	10
V	0	11

Table 5.6: Hashing Perfecto.

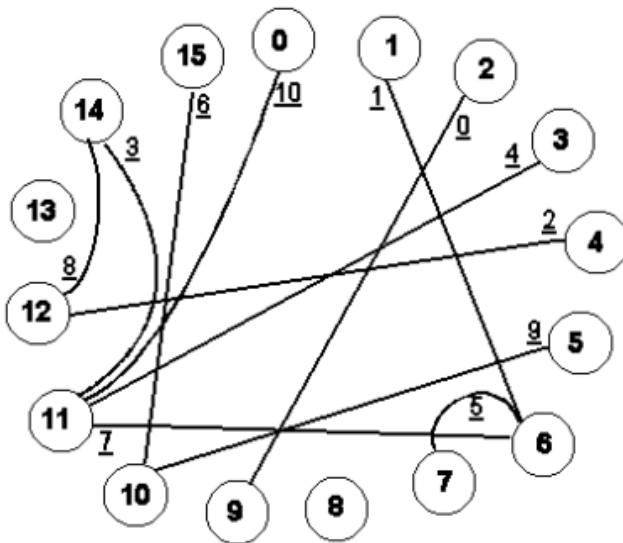


Figure 5.11: Hashing Perfecto Paso 1

El próximo paso es verificar que el grafo no tenga ciclos, si existen ciclos hay que probar con otras dos funciones de hashing de nuestra familia universal, para evitar tener que hacer este proceso muchas veces es importante que m sea un número bastante mas grande que n ya que en definitiva m es el número de nodos y n es el número de aristas en nuestro grafo y queremos minimizar la probabilidad de que el grafo tenga un ciclo.

Si el grafo no tiene ciclos es posible derivar la función de hashing perfecta, para hacerlo hay que rotular los vértices del grafo con un sencillo procedimiento. Comenzando desde un vértice cualquiera que no tenga rótulo lo rotulamos con 0 y luego vamos rotulando todos los vecinos de dicho vértice con la siguiente fórmula:

$$v = \text{arista} - \text{vertice} \pmod n$$

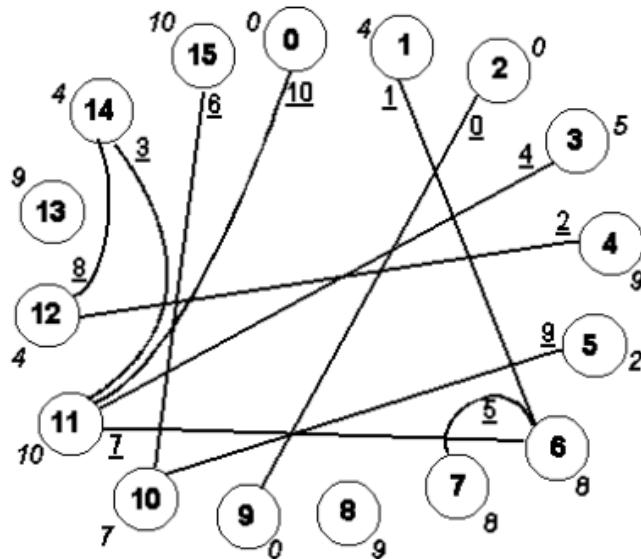


Figure 5.12: Hashing Perfecto Paso 2

Observemos en la figura 5.12 el resultado de rotular los vértices: empezamos con el vértice 0 y lo rotulamos con 0 luego para v_{11} calculamos $10-0=10 \pmod{11}$. Luego rotulamos a v_{14} como $3-10 \pmod{11} = 4$ y a v_3 como $4-10 \pmod{11} = 5$. Cuando no quedan mas vecinos por rotular si quedan vértices sin rotulo repetimos el proceso, es decir que hacemos este proceso por cada componente conexa del grafo.

Los rótulos de los m vértices del grafo definen una función auxiliar g que debemos almacenar en algún lado ya que formará parte de nuestra función de hashing perfecta.

A partir de la función g y de las dos funciones de hashing usadas h_1 y h_2 es posible construir una función de hashing perfecta para las claves usadas de la forma:

$$h(x) = g(h_1(x)) + g(h_2(x)) \pmod n$$

x	g(x)
0	0
1	4
2	0
3	5
4	9
5	2
6	8
7	8
8	9
9	0
10	7
11	10
12	4
13	9
14	4
15	10

Table 5.7: Hashing Perfecto función g.

Veamos como funciona en nuestro caso:

clave	h1(clave)	h2(clave)	g(h1)	g(h2)	$g(h1) + g(h2) \pmod n$
A	2	9	0	0	0
B	6	1	8	4	1
C	4	12	9	4	2
E	14	11	4	10	3
H	3	11	5	10	4
U	7	6	8	8	5
J	10	15	7	10	6
O	6	11	8	10	7
R	12	14	4	4	8
S	5	10	2	7	9
V	0	11	0	10	10

Table 5.8: Hashing Perfecto función final.

Observemos que la función generada es perfecta ya que a cada clave le asigna un único número entre 0 y 10 ($m = 11$), es también mínima ya que no hay ninguna posición en nuestra tabla final sin usar y finalmente preserva el orden de las claves originales lo cual a veces es un efecto colateral deseable.

Con este algoritmo es posible entonces construir una función de hashing perfecta, mínima y preservadora del orden. Para poder hacerlo es necesario conocer de antemano todas las claves que queremos almacenar en la tabla, generar la

función y almacenar g . Un detalle sumamente importante es que esta función no admite altas, bajas o modificaciones, es decir que solo tiene sentido cuando los datos son estáticos siendo esta la principal limitación del método.

5.9 The Hashing Trick

En esta sección vamos a explicar un mecanismo muy ingenioso e importante en Data Science conocido con varios nombres: "The Hashing Trick", "Feature Hashing" o "Hash Kernels". Vamos a explicar en primer lugar su funcionamiento desde el punto de vista práctico y en la siguiente sección desarrollaremos una explicación mas formal y teórica basada en el teorema de Johnson y Lindenstrauss.

Para entender el funcionamiento de THT vamos a recordar el modelo "Bag of Words" que habíamos explicado como una forma de representar textos. En BOW cada texto se representa mediante un vector que tiene tantas dimensiones como palabras existen en total en TODOS los documentos de la colección, en cada posición del vector indicamos la cantidad de veces que cada palabra ocurre en el documento. Como cada documento solo usa algunas palabras y no todas las palabras posibles estos vectores son terriblemente dispersos. En colecciones normales de texto estos vectores pueden llegar a tener cientos de miles o incluso millones de dimensiones y en algunos algoritmos esto es algo muy difícil de manejar. Mediante THT podemos reducir el tamaño de los vectores sin perder el poder de representación de los textos de nuestra colección.

Usando THT podemos decidir el tamaño de los vectores con los cuales queremos representar los documentos, una vez que hicimos esto por cada palabra del documento aplicamos una función de hashing que tiene como espacio de direcciones el tamaño de los vectores e incrementamos en 1 la posición indicada por la función de hashing. El vector final luego de procesar todas las palabras del texto es su representación.

THT tiene varias propiedades muy importantes e interesantes, en primer lugar es un método que preserva la naturaleza dispersa de los datos, esto es fácil de entender ya que solo encendemos las posiciones indicadas por la función de hashing para las palabras que encontramos en los textos, aun en pocas dimensiones esto genera vectores dispersos con los cuales es mas sencillo trabajar. Esto es mejor que una proyección aleatoria en donde el resultado serían siempre vectores densos.

THT no solo puede usarse para textos sino que con alguna variante u otra puede usarse como método genérico para representar cualquier tipo de dato, simplemente hay que decidir de que forma aplicar la función de hashing a cada columna de los datos. [Shi][Shi2]

Un tema a analizar es el efecto de las colisiones ya que puede ocurrir que dos "features" diferentes de nuestros datos originales colisionen en un mismo feature al representarlos mediante THT. Empíricamente se ha mostrado que el efecto de las colisiones es mínimo y en muchos casos es incluso favorable reduciendo el ruido del set de datos. Cuando el efecto de las colisiones no es bueno podemos



Figure 5.13: The Hashing Trick

usar el método de Weinberger para mitigarlas.

5.9.1 Método de Weinberger y Hash Kernels

En esta variante usaremos una segunda función de hashing que devuelve solamente dos valores posibles: -1 o +1 y usaremos este resultado para decidir si incrementamos o decrementamos la dimensión indicada por la primer función de hashing. Con este método el promedio de cada columna en nuestros vectores va a tender a cero, además se minimiza el efecto de las colisiones ya que no siempre incrementamos cuando dos features diferentes colisionan. El efecto mas interesante de este método es que el producto interno entre los vectores en nuestra nueva representación es equivalente al producto interno entre los vectores en el espacio original lo cual admite el uso de THT para crear Hash Kernels en donde usamos el producto interno entre los nuevos vectores para estimar la semejanza entre dos vectores (a mayor producto interno mas semejantes son los vectores).

Hay que destacar que en una gran cantidad de casos THT funciona muy bien sin necesidad de usar el método de Weinberger pero el mismo es necesario cuando queremos usar THT como un kernel. [Weinberger]

5.9.2 Filtros de Spam Personalizados

A modo de ejemplo comentaremos una aplicación directa de THT para la construcción de un filtro de Spam [Attenberg] que los usuarios pueden personalizar. El set de datos va a ser una colección de emails en donde marcaremos los que son Spam, el funcionamiento básico de nuestro filtro consiste en representar cada email con un vector y luego entrenar un clasificador lineal binario en base a los labels que tenemos de Spam o No-Spam, el clasificador aprende entonces a detectar si un mail es spam o no y podemos aplicarlo a los nuevos emails para predecir si son Spam.

Por supuesto la idea es usar THT para representar los textos en los emails por lo que podemos fijar el tamaño de nuestros vectores. El problema es como lograr que cada usuario pueda personalizar el algoritmo en base a lo que cada usuario considera Spam o no y al mismo tiempo mantener un cierto filtro global en el cual podamos aprovechar el feedback de todos los usuarios al mismo tiempo.

La solución usando THT es muy sencilla y elegante. Cada vez que un usuario marca un mail como Spam o No procesamos cada palabra y aplicamos una función de hashing a la palabra pero también a la concatenación del usuario y la palabra. Es decir que si la palabra es "perro" y el usuario es 19931 hashearemos $h(\text{perro})$ y $h(19931\text{perro})$. En el vector incrementaremos ambas posiciones por lo que en un mismo vector van a coexistir datos globales y datos personalizados.

De esta forma podemos entrenar un único clasificador para todos los usuarios de forma tal que el feedback de todos los usuarios aporta al clasificador global pero a su vez el feedback de cada usuario aporta también a que su filtro sea personalizado. Lo mas notable de este método es que las colisiones en los vectores no solo no tienen un efecto perjudicial sino que mejoran la performance del clasificador.

La clave está en que cuando queremos predecir si un mail es spam o no para un determinado usuario vamos a hashear cada palabra del mail y al mismo tiempo la concatenación del usuario y cada palabra. Luego con el vector obtenido aplicamos el algoritmo de clasificación que hemos entrenado y este nos dirá si es Spam o No. Una característica muy interesante de este sistema es que si algunos usuarios marcan muchos mails como Spam y otros marcan muy pocos el filtro de Spam no tiene un rendimiento diferente para ambos ya que se aprovecha lo marcado por todos los usuarios! En el caso del usuario que marca pocos mails como Spam su filtro estará simplemente menos personalizado.

5.10 El Teorema de Johnson y Lindenstrauss

Este teorema es fundamental en Data Science ya que por primera vez vamos a poder justificar en forma teórica que es posible trabajar con un set de datos en menos dimensiones que las que tiene originalmente. Veamos lo que dice el teorema:

Theorem 8 (Johnson-Lindenstrauss) *Para todo conjunto de n datos en d dimensiones y una cierta tasa de error $0 < \epsilon < 1$ existe un entero positivo k de*

la forma:

$$k = 4 \frac{1}{\frac{\epsilon^2}{2} - \frac{\epsilon^3}{3}} \log(n)$$

Y existe una proyección $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$ que cumple:

$$(1 - \epsilon) \|u - v\| \leq \|f(u) - f(v)\| \leq (1 + \epsilon) \|u - v\|$$

Lo que el teorema nos dice, en definitiva es que es posible representar a los datos en una cantidad de dimensiones en el orden de $O(\log(n))$ siendo n la cantidad de datos de forma tal que las distancias en el nuevo espacio dimensional sean muy similares a las distancias entre los puntos en el espacio original.

A modo de ejemplo veamos según el error ϵ y la cantidad de puntos cual sería el valor de k según el teorema.

	$\epsilon = 0.2$	$\epsilon = 0.1$	$\epsilon = 0.01$
n=1000	1595	5921	556330
n=100000	2657	9869	927216
n=1000000	3189	11842	1112659

Table 5.9: Johnson Lindestrauss.

Podemos ver, por ejemplo que si tenemos un millón de puntos y queremos una tasa de error del 10% podemos representar a nuestros datos en 11842 dimensiones. Esto puede parecer mucho pero hay que tener en cuenta que podemos pasar de cualquier cantidad de dimensiones a 11842 con un error mínimo para las distancias entre los puntos, por ejemplo podríamos tener originalmente un millón de dimensiones y el teorema de JL nos indicaría que podemos proyectar ese millón de dimensiones en 11842 manteniendo las distancias entre los puntos a lo sumo a un 10% de las distancias originales.

Hasta el momento el teorema de JL nos informa la existencia de una proyección pero no nos dice como generarla, vamos a ver aquí dos métodos muy simples que permiten generar una matriz de proyección que cumple con el teorema de JL.

5.10.1 Primera Proyección

La primera proyección de JL es muy simple: una matriz de $A = dxk$ en donde cada elemento A_{ij} de la matriz surge de una distribución normal con promedio 0 y desviación standard 1 $\mathcal{N}(0, 1)$

Por ejemplo:

$$\begin{pmatrix} -1.1813 & -1.4869 & -1.1998 \\ -0.2834 & 0.0070 & -0.9376 \\ -0.2118 & 0.4475 & -0.6213 \\ -1.2673 & 1.3946 & 0.6132 \\ -0.2562 & -1.0749 & -0.2116 \end{pmatrix}$$

5.10.2 Segunda Proyección

Si partimos de un conjunto de n datos en d dimensiones y queremos proyectarlos a k dimensiones nuestra segunda matriz de proyección es una matriz de $d \times k$ en donde cada elemento r_{ij} de la matriz es:

$$r_{Ij} = \begin{cases} +1, & \text{con probabilidad } 1/2. \\ -1, & \text{con probabilidad } 1/2. \end{cases} \quad (5.2)$$

Ejemplo para proyectar de 5 a 3 dimensiones:

$$\begin{pmatrix} +1 & -1 & -1 \\ +1 & +1 & +1 \\ -1 & +1 & -1 \\ +1 & +1 & -1 \\ -1 & -1 & +1 \end{pmatrix}$$

Para realizar la proyección se calcula:

$$x' = \frac{x * M}{\sqrt{k}}$$

Es decir que simplemente multiplicamos los datos por la matriz de proyección y escalamos por la raíz de k

5.10.3 Tercera Proyección

La tercera proyección define una matriz dispersa en donde $2/3$ de sus elementos son ceros y los restantes se distribuyen equitativamente entre $+1$ y -1 .

$$r_{Ij} = \sqrt{3} * \begin{cases} +1, & \text{con probabilidad } 1/6. \\ 0, & \text{con probabilidad } 2/3. \\ -1, & \text{con probabilidad } 1/6. \end{cases} \quad (5.3)$$

Ambas proyecciones se describen en mayor detalle en [Achlioptas]. La ventaja de la tercera proyección con respecto a la segunda o la primera es que es computacionalmente mas eficiente para efectuar los cálculos.

5.10.4 Por qué funciona el Teorema de JL

Hemos visto el enunciado del teorema y también tres matrices de proyección que sabemos que cumplen con el mismo. Vamos a demostrar aquí que estas matrices de proyección sirven para preservar las normas de los vectores, es decir que las normas de los vectores proyectados se aproximan a las normas de los vectores originales por un cierto factor de escala.

Demostrar que una matriz de proyección preserva las normas automáticamente implica que la misma preserva las distancias entre los vectores ya que la distancia entre v_1 y v_2 es la norma de la diferencia $\|v_1 - v_2\|$. De la misma

forma podemos demostrar que preservando las normas se preservan los productos internos entre los vectores. Esto implica que a partir de este punto solo nos interesa demostrar la preservación de las normas y esto automáticamente nos garantiza que también se preserven las distancias y los productos internos.

El motivo por el cual se preservan las normas se debe al fenómeno conocido como *concentration of measures* que nos dice que la norma de un vector está fuertemente concentrada en su promedio, esto implica que si hacemos una proyección aleatoria de un vector y la elevamos al cuadrado el resultado approxima la norma del vector al cuadrado. Esto es lo que vamos a demostrar a continuación.

Theorem 9 (Concentration of Measures) *Sea x un vector de $d \times 1$ y sea A una matriz de proyección de $m \times d$ con $m < d$ en donde cada elemento de A A_{ij} surge o bien de una distribución Gaussiana con promedio 0 y desviación standard 1 o bien es +1 o -1 de forma equiprobable. Sea v_m el resultado de Ax .*

Queremos demostrar que:

$$\mathbb{E}[[Ax]_j^2] = \|x\|^2 \quad (5.4)$$

Y además

$$\mathbb{E}\left[\frac{1}{\sqrt{m}}\|Ax\|\right] = \|x\| \quad (5.5)$$

Para demostrar la primera parte vamos a considerar el valor de $[Ax]_j^2$ es decir una fila de la matriz A de $m \times d$ que es un vector de d elementos.

$$\begin{aligned} \mathbb{E}[[Ax]_j^2] &= \mathbb{E}\left[\left(\sum_{i=1}^d A_{ij}x_i\right)^2\right] \\ &= \mathbb{E}\left[\sum_{i,i'} A_{ij}A_{i',j}x_i x_{i'}\right] \\ &= \mathbb{E}\left[\sum_i A_{i,j}^2 x_i^2\right] \\ &= \sum_i x_i^2 \\ &= \|x\|^2 \end{aligned} \quad (5.6)$$

Es decir que cada uno de los m elementos de Ax al cuadrado tiene como valor esperado la norma de x al cuadrado. Si queremos calcular ahora la norma de Ax partimos de que $\|Ax\|_j^2 = \|x\|^2$ y como A es de $m \times d$ tenemos que

multiplicar por m . Nos queda: $\|Ax\|^2 = m\|x\|^2$ tomando la raíz en ambos lados: $\|Ax\| = \sqrt{m}\|x\|$ por lo tanto:

$$\frac{1}{\sqrt{m}}\|Ax\| = \|x\| \quad (5.7)$$

Que es la segunda parte que queríamos demostrar (siempre considerando el valor esperado).

El teorema queda demostrado bajo la única condición de que los elementos de A son independientes y tienen varianza=1. Es decir que esto vale tanto para la primera como la segunda proyección de JL, ya que en ambos casos vale la misma demostración.

Notemos que el teorema de concentration of measures nos dice que el cuadrado de cada proyección tiene como valor esperado el cuadrado de la norma del vector original, esto implica, en principio que cada proyección tiene como valor esperado la norma del vector, sin embargo es importante mantener los cuadrados ya que mas adelante los vamos a necesitar.

Si tenemos entonces un vector v_d de d dimensiones y usamos una matriz de proyección A como las que describimos de $m \times d$ para obtener el vector proyectado $v_m = Av_d$ entonces:

$$\frac{1}{\sqrt{m}}\|v_m\| = \|v_d\| \quad (5.8)$$

Esto quiere decir que las matrices descriptas anteriormente preservan la norma de los vectores originales con un cierto factor de escala.

Esto aplica tanto para la matriz Gaussiana como para la matriz formada por +1 y -1 de forma equiprobable. Queremos ver que ocurre si reducimos la cantidad de +1 o -1 en la matriz ya que esto nos permitiría usar una matriz de proyección que no es densa y por lo tanto sería computacionalmente mas eficiente.

Theorem 10 *Sea A una matriz de $d \times m$ en donde cada fila de la matriz tiene k elementos +1 o -1 y el resto son ceros. Queremos demostrar que esta matriz preserva las normas de los vectores de d dimensiones al proyectarlos en m dimensiones con un cierto factor de escala.*

$$\|v_k\| \approx \sqrt{k}\|v_d\| \quad (5.9)$$

Por ejemplo si queremos pasar de 5 a 3 dimensiones podemos usar:

$$\begin{pmatrix} +1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & +1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & 0 \end{pmatrix}$$

Nota importante: Si los vectores son de $1 \times d$ entonces la matriz de proyección es de $d \times m$ y tiene k +1 o -1 por fila. Si los vectores son de $d \times 1$ entonces la

matriz de proyección es de $m \times d$ y cada columna de la matriz tiene $k+1$ o -1 . Es decir la matriz transpuesta. A partir de aquí podemos usar indistintamente una u otra notación.

Para demostrar este teorema vamos a partir de que sabemos que si tomamos un vector v_d de d dimensiones y le aplicamos una proyección aleatoria que consiste de $d+1$ o -1 lo que obtenemos al cuadrado tiene como valor esperado la norma del vector al cuadrado.

Observemos que cada columna de nuestra matriz de $d \times m$ equivale a una proyección aleatoria de d elementos, por lo tanto cuando multiplicamos $v_d * A$ lo que estamos haciendo es obteniendo m proyecciones aleatorias. Pero nuestra matriz no tiene $d+1$ o -1 en cada columna. La cantidad total de $+1$ o -1 en nuestra matriz es $d*k$ ya que tenemos $k+1$ o -1 por fila y un total de d filas. Por lo tanto la cantidad promedio de elementos distintos de cero en cada columna de nuestra matriz es $\frac{dk}{m}$.

Sabemos que con $d+1$ o -1 el cuadrado de la proyección tiene como valor esperado el cuadrado de la norma. Por lo tanto si tenemos $\frac{k}{m}+1$ o -1 el cuadrado de la proyección tiene como valor esperado $\frac{k}{m}\|x_d\|^2$

$$v_{mi}^2 = \frac{k}{m}\|x_d\|^2 \quad (5.10)$$

Podemos ahora calcular el valor esperado de la norma de v_m de la forma:

$$\begin{aligned} v_{mi}^2 &= \frac{k}{m}\|x_d\|^2 \\ \sum_i v_{mi}^2 &= m \frac{k}{m}\|x_d\|^2 \\ \sqrt{\sum_i v_{mi}^2} &= \sqrt{m \frac{k}{m}\|x_d\|^2} \\ \|v_m\| &= \sqrt{k}\|x_d\| \end{aligned} \quad (5.11)$$

Lo cual demuestra el teorema.

$$\|v_m\| = \sqrt{k}\|x_d\| \quad (5.12)$$

$$\|v_d\| = \frac{1}{\sqrt{k}}\|x_m\| \quad (5.13)$$

Cuando todos los elementos de la matriz son $+1$ o -1 , que es la segunda proyección de JL tenemos que $k = m$ por lo tanto $\frac{1}{\sqrt{m}}\|x_m\| = \|x_d\|$ que es lo que demostramos en el teorema de concentration of measures. Y si $k = 1$ tenemos que $\|v_m\| = \|v_d\|$ es decir que la proyección aproxima el valor de las normas del espacio original con factor de escala igual a uno (!).

Esto quiere decir que una matriz de $d \times m$ en donde cada fila tiene k elementos +1 o -1 sirve para proyectar los vectores de d a m dimensiones preservando las normas entre los mismos con factor $\frac{1}{\sqrt{k}}$, el factor de escala no es lo importante sino que la matriz preserva las normas. Este tipo de matrices van a ser fundamentales para mostrar la relación entre "The Hashing Trick" y el teorema de JL.

Finalmente y como ya hemos mencionado recordemos que preservar las normas implica preservar las distancias y también los productos internos entre los vectores.

5.10.5 The Hashing Trick y el Teorema de Johnson Lindenstrauss

Existe una relación directa entre esta matriz y el THT de acuerdo al método de Weinberger. Consideremos el siguiente ejemplo en donde damos el valor de las dos funciones de hashing para aplicar THT según el método de Weinberger para convertir datos de 10 dimensiones a 4.

Dimension	h1(dim)	h2(dim)
0	2	+1
1	0	-1
2	1	-1
3	3	+1
4	1	+1
5	2	-1
6	0	-1
7	3	+1
8	2	+1
9	0	-1

Table 5.10: Johnson Lindestrauss y THT.

Por ejemplo si nuestro vector original fuera: $x = (3, 0, 0, 4, 1, 2, 1, 0, 1, 3)$ aplicando THT quedaría $x' = (-4, 1, -1, 7)$

Observemos ahora que a partir de las funciones de hashing definidas por THT podemos crear una matriz de proyección. Por ejemplo que para la dimensión 0 la función de hashing 1 nos da 3 y la segunda +1 equivale a poner un +1 en la fila 0 y columna 3 de nuestra matriz de proyección que quedaría de la siguiente forma:

$$\begin{pmatrix} 0 & 0 & +1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & +1 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & +1 \\ 0 & 0 & +1 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$$

Notemos que usar THT es equivalente a multiplicar un vector por esta matriz. La cantidad de funciones de hashing a usar incide en la cantidad de +1 o -1 que tendremos por cada fila. Esta matriz es la matriz que hemos demostrado que preserva las normas y por lo tanto podemos afirmar que The Hashing Trick preserva las normas y por lo tanto las distancias y los productos internos entre los vectores. Esto es revelador y terriblemente importante ya que justifica el uso de hashing para reducir la dimensionalidad de los datos.

Observemos que las colisiones pueden causar que la matriz tenga elementos que no son +1 o -1 por ejemplo si colisionan tres +1 tendremos un 3 en la matriz. En general esto no afecta la validez del teorema ya que la suma de elementos de la matriz sigue teniendo como valor esperado promedio 0.

La cantidad de funciones de hashing y la cantidad de datos influyen en la tasa de error que tendremos para las normas, distancias y productos internos. Si lo que queremos es saber cuáles son los vecinos mas cercanos a un cierto punto no nos interesa realmente el valor numérico de la distancia sino simplemente que las relaciones que había en el espacio original se preserven en el nuevo espacio y hemos mostrado que el hashing trick cumple con esta propiedad.

Chapter 6

LSH

Any sufficiently advanced
technology is indistinguishable
from magic

—Arthur C. Clarke

LSH (locality sensitive hashing) es una solución muy eficiente al problema de los vecinos mas cercanos (nearest neighbors). En el primer capítulo desarrollamos este problema y vimos que la solución por fuerza bruta es muy ineficiente ya que por cada punto tiene que calcular la distancia contra todos los demás puntos para encontrar el mas cercano o los k mas cercanos. Este problema aplica no solo a algoritmos como KNN sino a una enorme cantidad de otros algoritmos que necesitan encontrar el punto mas cercano o los k puntos mas cercanos a un cierto punto de nuestro set de datos. LSH nos brinda una solución aproximada a este problema pero de forma mucho mas rápida que la búsqueda por fuerza bruta.

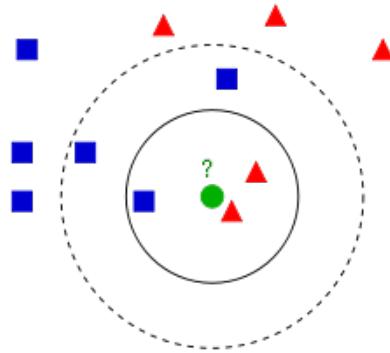


Figure 6.1: El problema de los vecinos mas cercanos

El concepto detrás de LSH es muy simple: aplicar a los puntos una función de hashing de forma tal que los puntos que son parecidos (cercanos) colisionen. Si logramos construir esta función es entonces sencillo resolver el problema de los vecinos mas cercanos, dado un punto "query" aplicamos la función de hashing y accedemos al bucket de la tabla de hash indicado por la misma, luego calculamos la distancia entre nuestro punto "q" y todos los puntos dentro del bucket devolviendo el más cercano o los k mas cercanos según el caso.

Nuestra función de hashing puede procesar cualquier punto en \mathbb{R}^d dimensiones y nos da como resultado un número de bucket, nos interesa la distribución de probabilidades que la función LSH hace sobre el conjunto de buckets. Supongamos que queremos encontrar los puntos mas cercanos a un cierto punto query q , queremos que la función LSH tenga una alta probabilidad de encontrar buckets con puntos cercanos y una probabilidad baja de encontrar buckets con puntos lejanos. Necesitamos entonces dos distancias y dos probabilidades. Llamaremos d_1 a la distancia para la cual queremos que dos puntos sean similares, y llamaremos p_1 a la probabilidad de que la función LSH nos de una colisión si la distancia entre dos puntos es menor o igual a d_1 . Por otro lado llamaremos d_2 a la distancia para la cual queremos que dos puntos no sean similares y será p_2 la probabilidad de que la función nos de una colisión si los puntos están a distancia p_2 o mas.

$$\text{LSH}(d_1, d_2, p_1, p_2) \quad (6.1)$$

En general fijamos $d_2 = c * d_1$ para una cierta constante c y queremos que $p_1 > p_2$, cuanto mayor es la diferencia entre p_1 y p_2 mejor será la función LSH. Una forma de evaluar esto es usando:

$$\rho = \frac{\log 1/p_1}{\log 1/p_2} \quad (6.2)$$

De esta forma podemos comparar diferentes funciones LSH buscando maximizar el valor de ρ de forma empírica.

Para poder crear una función de hashing LSH es necesario saber con qué métrica (distancia) se quiere trabajar, en las próximas secciones mostramos como crear funciones LSH para diferentes distancias.

6.1 LSH para la distancia de Jaccard

En esta sección desarrollaremos una función LSH para documentos de texto basada en la distancia de Jaccard. Para empezar vamos a procesar cada documento obteniendo todos los n-gramas de n caracteres a los cuales denominaremos "shingles".

Por ejemplo para "hola" usando bigramas los shingles son \$h,ho,ol,la,a\$. Si usamos trigramas entonces \$ho,hol,ola,la\$. La elección de n para el tamaño de los n-gramas depende de los datos, para documentos de texto en general un valor de n entre 5 y 10 caracteres es una buena idea.

La semejanza de Jaccard entre dos documentos se define como:

$$sJ(D1, D2) = \frac{D1 \cap D2}{D1 \cup D2}$$

Es decir que la semejanza es el cociente entre la cantidad de shingles que los documentos tienen en común (intersección) y la cantidad total de shingles en ambos documentos (unión).

La semejanza de Jaccard nos da un número entre 0 y 1, vale 0 cuando los documentos no tienen absolutamente ningún shingle en común y vale 1 cuando los documentos tienen exactamente los mismos shingles. En base a esto podemos definir la distancia de Jaccard como:

$$dJ(D1, D2) = 1 - \frac{D1 \cap D2}{D1 \cup D2}$$

Es importante tener en cuenta cuando estamos hablando de distancia y cuando de semejanza recordando que uno es 1 menos el otro.

Imaginemos ahora una matriz en la cual las filas son todos los shingles que existen en nuestra colección y cada columna representa un documento, indicando con 1 o 0 según el shingle aparezca o no en el documento.

shingle	d1	d2	d3	d4
s0	0	1	0	0
s1	0	0	1	1
s2	1	1	0	1
s3	0	1	0	0
s4	0	0	1	0

Table 6.1: Shingles y Documentos.

Esta matriz no hace falta construirla realmente, la vamos a usar única y exclusivamente con fines teóricos. Tomemos ahora una permutación al azar de las filas de la matriz y consideremos en qué número de fila aparece el primer 1 en cada columna, a este número lo llamaremos "minhash" (MH) cada documento tiene entonces un minhash asociado.

shingle	d1	d2	d3	d4
s3	0	1	0	0
s0	0	1	0	0
s2	1	1	0	1
s4	0	0	1	0
s1	0	0	1	1
MH	2	0	3	2

Table 6.2: Shingles y Documentos.

Queremos demostrar que la probabilidad de que el minhash de dos documentos coincida es igual a la semejanza de Jaccard entre ambos documentos.

Para demostrar esto vamos a considerar dos documentos aislados cualesquiera D1 y D2. Existen 3 posibilidades para cada shingle:

1. El shingle aparece en ambos documentos (11 en la matriz)
2. El shingle no aparece en ninguno de los documentos (00 en la matriz)
3. El shingle aparece en uno de los documentos pero no en el otro (10 o 01 en la matriz)

Llamemos x a la cantidad de casos en los cuales el shingle aparece en ambos documentos y y a los casos en los cuales el shingle aparece en un documento o en el otro pero no en ambos.

Para calcular la probabilidad de que el primer 1 de cada columna coincida tenemos que los casos favorables son aquellos en los cuales el shingle aparece en ambos documentos: x y los casos totales son aquellos en los cuales el shingle aparece en ambos documentos o bien aparece en uno pero no en el otro. Los casos en los cuales el shingle no aparece en ningún documento no hace falta contarlos porque son dos ceros y no afectan el lugar en donde aparece el primer 1.

$$P[MH(D1) = MH(D2)] = \frac{x}{x+y}$$

Observemos ahora que esta probabilidad es igual a la semejanza de Jaccard entre los documentos ya que para la semejanza de Jaccard tomamos la intersección es decir aquellos shingles que aparecen en ambos documentos que sabemos que es x y dividimos por la unión siendo la unión la suma de aquellos shingles que aparecen en ambos documentos (x) y los shingles que aparecen en uno u otro documento (y). Notar que aquellos shingles que no aparecen en ninguno de los dos documentos no forman parte ni de la unión ni de la intersección

$$sJ(D1, D2) = \frac{x}{x+y}$$

Como podemos ver la semejanza de Jaccard entre dos documentos es igual a la probabilidad de que los minhashes de ambos documentos coincidan.

$$P[MH(D1) = MH(D2)] = sJ(D1, D2)$$

De aquí se desprende que comparar los minhashes de los documentos es una forma lógica de evaluar la semejanza entre ambos. Notemos también que un solo minhash nos da muy poca granularidad para medir la semejanza entre dos documentos, si solo tenemos un minhash nuestras opciones se limitan a decir que los documentos son similares (el minhash coincide) o bien no lo son (el minhash no coincide).

La solución, por supuesto es usar mas de un minhash y luego analizar cuantos minhash coinciden usando esta medida como semejanza entre documentos. Si usamos 4 minhash tenemos que dos documentos pueden tener semejanza 0, 1/4, 1/2, 3/4 o 1. Según coincidan 0,1,2,3 o los 4 minhash.

Veamos entonces de que forma podemos calcular el minhash de un documento de forma eficiente.

Vamos a usar tantas funciones de hashing como minhash queremos usar y vamos a construir una pequeña matriz que tiene tantas filas como documentos y tantas columnas como minhashes. Inicialmente el minhash de cada documento lo inicializaremos en infinito.

Ejemplo:

	mh1	mh2	mh3	mh4
d1	∞	∞	∞	∞
d2	∞	∞	∞	∞
d2	∞	∞	∞	∞

Table 6.3: Minhashes.

A continuación vamos a procesar documento por documento, extrayendo los shingles y hasheando cada shingle con cada una de las 4 funciones de hashing. Supongamos que para el primer shingle de nuestro documento las funciones de hashing nos devuelven 23,317,41 y 1145 respectivamente. Lo que hacemos es actualizar la tabla siempre y cuando el valor de la función de hashing sea menor al valor existente en la tabla.

	mh1	mh2	mh3	mh4
d1	23	317	41	1145
d2	∞	∞	∞	∞
d2	∞	∞	∞	∞

Table 6.4: Minhashes.

Este proceso continua con el siguiente shingle del primer documento, luego pasamos al documento 2, y repetimos hasta haber procesado todos los documentos. En una sola lectura secuencial de nuestra colección de textos podemos obtener el valor de todos los minhashes que deseemos para cada documento sin necesidad de espacio especial en memoria.

Tenemos que mostrar ahora que el minhash calculado de esta manera es equivalente a la posición del primer uno en la columna de nuestra matriz de shingles y documentos para probar que sigue siendo válido que la probabilidad de coincidencia entre dos minhash sigue siendo igual a la semejanza de Jaccard entre los documentos.

La intuición para demostrar esto es muy simple: el valor que nos devuelve la función de hashing para un shingle lo podemos considerar igual al número de

fila en la cual aparecería dicho shingle en una permutación al azar de la matriz. Por lo tanto el valor mínimo de minhash es equivalente al número de fila en el cual aparecería el primer uno de la matriz. Notar que los shingles que NO aparecen en un documento directamente no se hashean. De esto se desprende que el espacio de direcciones de las funciones de hashing a usar para los minhash es igual o mayor a la cantidad total de shingles en nuestra colección.

Hemos presentado entonces un algoritmo muy simple para calcular cualquier cantidad de minhash a partir de una colección de documentos y hemos demostrado que la probabilidad de que dos minhash coincidan es igual a la semejanza de Jaccard entre los documentos. Con estos elementos vamos a construir nuestra primera función de hashing LSH. En la siguiente sección analizaremos el efecto que tenemos al usar mas de un minhash y como construir la función LSH a partir de varios minhash.

6.2 Amplificación de funciones LSH

Antes de desarrollar de forma teórica el uso de varios minhash al mismo tiempo vamos a hacer una aproximación intuitiva al problema analizando el efecto de agrupar los minhash de diferentes formas.

Supongamos que usamos un solo minhash, cada documento está representado entonces por un único número. Cuando queremos analizar los documentos similares a uno dado buscamos aquellos que coincidan con el minhash. Como usamos un solo minhash es posible que devolvamos documentos que no son realmente similares al que buscamos (falsos positivos) y también puede pasar que algún documento que si es parecido no haya obtenido el mismo minhash por lo que nuestra consulta lo ignoraría (falsos negativos).

Para reducir los falsos positivos podemos usar mas de un minhash y decir que dos documentos son similares si coinciden todos sus minhashes, cuantos mas minhashes usemos menos probable va a ser que tengamos falsos positivos. Es evidente que al mismo tiempo que reducimos la cantidad de falsos positivo estamos aumentando la cantidad de falsos negativos, con una cantidad muy grande de minhashes puede darse que no encontremos absolutamente ningún documento similar a otro.

Una forma de solucionar este problema es dividir a los minhashes en b grupos de r minhashes cada grupo. Por ejemplo si usamos 12 minhashes los podemos dividir en 4 grupos de 3 minhashes cada uno. A la hora de evaluar una consulta vamos a decir que un documento es similar a otro si en alguno de los 4 grupos los 3 minhashes coinciden. Esto es equivalente a hacer un OR de 4 construcciones AND de 3 minhashes cada una.

Si agrupamos los 4 minhashes en 2 grupos de 2 minhashes cada uno entonces d1 y d3 son similares ya que para el primer grupo de minhashes (mh1 y mh2) sus valores coinciden y d2 y d3 también son similares ya que para el segundo grupo (mh3, mh4) sus valores coinciden.

Habiendo realizado la explicación de como funciona agrupar minhashes para disminuir la cantidad de falsos positivos y falsos negativos vamos a desarrollar

	mh1	mh2	mh3	mh4
d1	3	17	4	11
d2	1	0	25	2
d3	3	17	25	2

Table 6.5: Agrupando Minhashes.

la teoría detrás de todo esto.

Lo que queremos lograr es dado un cierto umbral de semejanza de Jaccard que nuestra función encuentre los documentos si su semejanza es igual o mayor al umbral pero no los encuentre si es menor. Gráficamente es lo siguiente (fig 5.2)

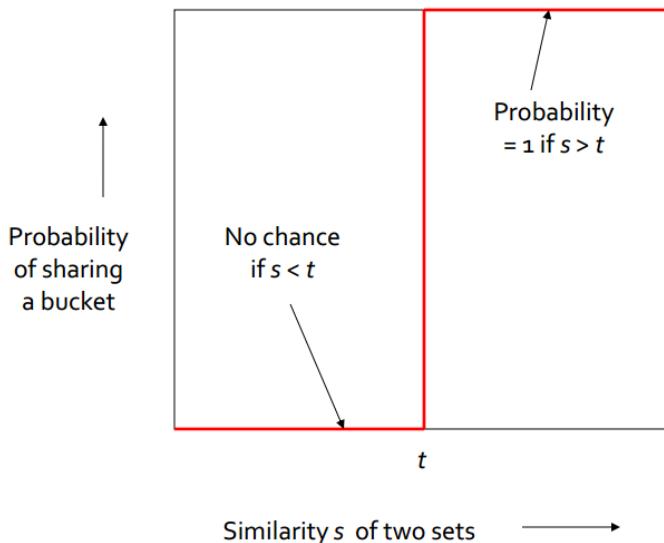


Figure 6.2: LSH caso ideal

El eje x indica la semejanza de Jaccard entre dos documentos, el eje y indica la probabilidad de que los documentos sean candidatos es decir que nuestra función los encuentre.

Si usamos un solo minhash nuestro gráfico tiene el siguiente aspecto:

Tenemos una línea ya que la probabilidad de que dos documentos sean candidatos es igual a la semejanza de Jaccard entre ambos, recordemos que la probabilidad de que los minhashes coincidan es igual a la semejanza de Jaccard por lo tanto usando solo un minhash la probabilidad de que dos documentos sean candidatos es igual a la semejanza de Jaccard entre ambos. Destaquemos

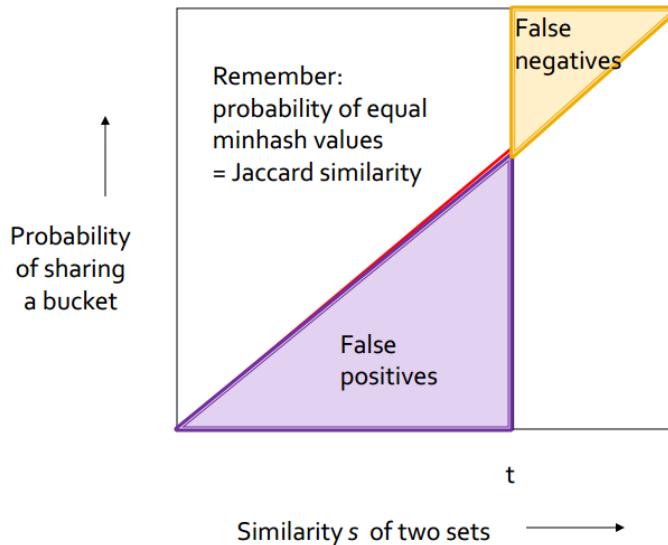


Figure 6.3: LSH con un solo minhash

cual es la diferencia con el caso ideal. En el caso ideal fijamos una semejanza por ejemplo 0.7 y queremos que los documentos sean candidatos si su semejanza es 0.7 o mayor y que no sean encontrados si su semejanza es menor a 0.7. Usando solo un minhash si dos documentos tienen semejanza 0.7 la probabilidad de encontrarlos es 0.7 y queremos que sea mayor, cercana a 1. Por otro lado si dos documentos tienen semejanza 0.5 la probabilidad de que sean candidatos es 0.5 y queremos que sea mucho menor, cercana a 0 ya que 0.5 es menor a nuestro umbral de 0.7.

Vamos a calcular ahora cual es la probabilidad de que dos documentos sean candidatos usando b bandas (grupos) de r minhashes cada una.

Llaremos p a la probabilidad de que dos minhashes coincidan, es decir que p es igual a la semejanza de Jaccard entre los documentos. Si usamos r minhashes y pedimos que todos coincidan para que los documentos sean candidatos entonces la probabilidad de que lo sean es p^r . La probabilidad de que NO coincidan es entonces $1 - p^r$

Si tenemos b bandas (grupos) de r funciones de hashing cada una. La probabilidad de que NO haya coincidencia en una banda es $1 - p^r$ y la probabilidad de que no haya coincidencia en ninguna banda es entonces $(1 - p^r)^b$ por lo tanto la probabilidad de que haya coincidencia en ALGUNA banda es:

$$1 - (1 - p^r)^b$$

Esta fórmula es muy importante ya que nos permite calcular la probabilidad

de que dos documentos sean candidatos partiendo de una cierta semejanza de Jaccard usando una cierta cantidad de bandas b de r minhashes cada una, es decir un total de $b * r$ minhashes.

Graficando la fórmula tenemos algo así:

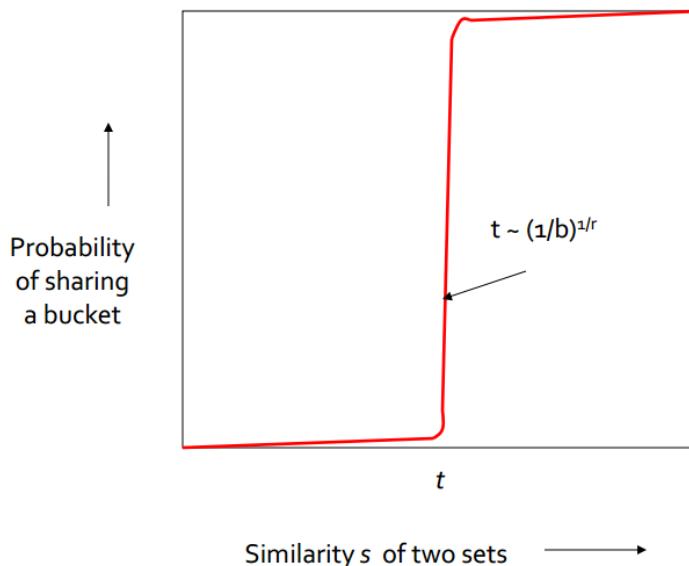


Figure 6.4: LSH con varios minhash

Vemos que se acerca a lo que buscamos el efecto de aumentar r disminuye los falsos positivos:

Mientras que aumentar b disminuye los falsos negativos.

Ejemplo: Estudiemos ahora un caso práctico en el cual queremos que nuestros documentos sean semejantes si su semejanza de Jaccard es 0.8 o más y que no sean semejantes si su semejanza es 0.6 o menor. Analicemos cuantas funciones de hashing hacen falta y de qué forma distribuirlas.

Para encontrar el resultado iremos probando valores de b y r comenzando con $b = r$ a modo de tanteo, y calcularemos la probabilidad de que sean candidatos para $p = 0.8$ y $p = 0.6$ nuestro objetivo es que la primera sea cercana a 1 y la segunda cercana a cero.

Una vez que llegamos a valores similares a los que necesitamos podemos ir variando la proporción de r y b hasta llegar al resultado que queremos. Por ejemplo usando $b = 8$ y $r = 6$ es decir un total de 48 funciones de hashing logramos que la probabilidad cuando la semejanza es 0.8 sea 0.9 y que la probabilidad cuando la semejanza es 0.6 baje a 0.2.

Cuanto más cercanas estén las semejanzas más funciones de hashing vamos a necesitar, por ejemplo si queremos que documentos que tienen semejanza 0.8

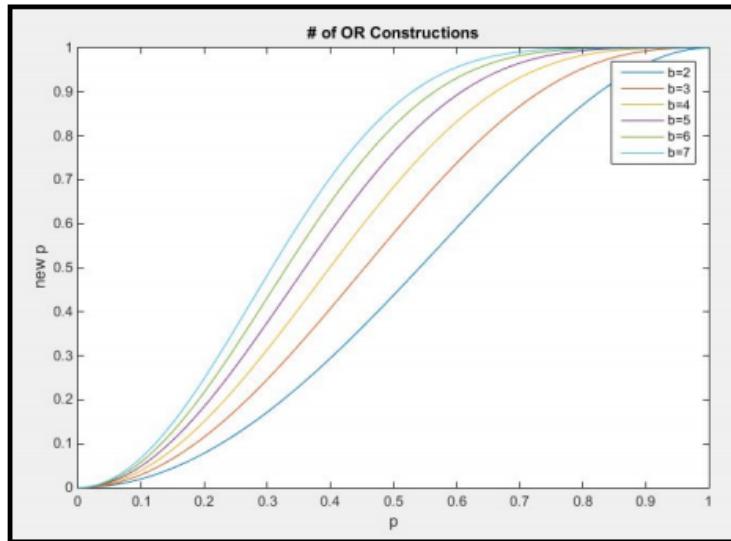


Figure 6.5: Efecto de aumentar r

b	r	$1 - (1 - 0.8^r)^b$	$1 - (1 - 0.6^r)^b$
1	1	0.8	0.6
2	2	0.8704	0.5904
3	3	0.8838	0.5181
4	4	0.8785	0.4260

Table 6.6: Buscando b y r.

sean candidatos pero documentos que tienen semejanza 0.75 no lo sean podemos necesitar miles de minhashes para lograrlo.

6.3 La función LSH en la práctica

Antes de continuar con el desarrollo teórico es conveniente realizar una pausa para explicar de que forma se construye y se usa una función LSH con las herramientas que hemos visto hasta el momento. Supongamos a efectos prácticos que hemos decidido usar 24 minhashes en 6 grupos de 4 minhashes cada uno ($r = 4, b = 6$).

La construcción de los minhashes ya la hemos estudiado por lo que luego de procesar secuencialmente todos los documentos de nuestra colección tendremos para cada documento sus 24 minhashes.

Como tenemos 6 grupos vamos a construir 6 tablas de hash, cada una de

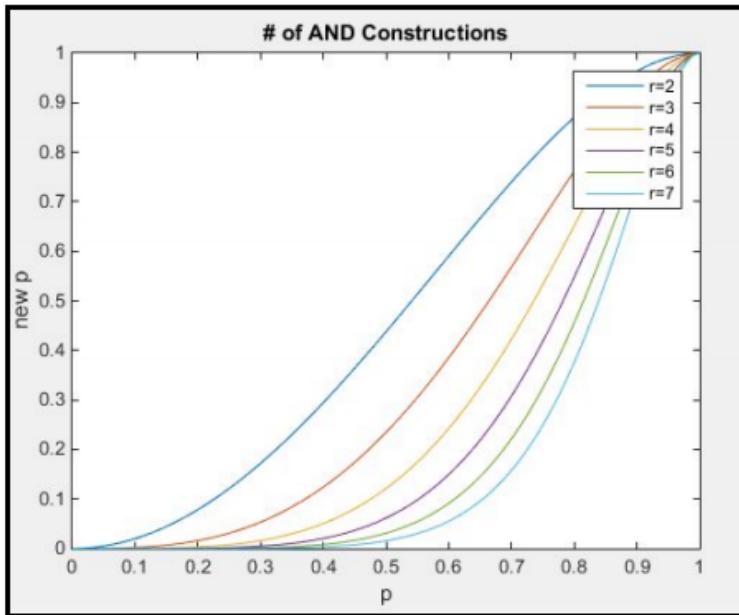


Figure 6.6: Efecto de aumentar b

una cierta cantidad de posiciones a definir, por ejemplo 1 millón. Cada posición de esta tabla guardará una lista o hash de cuales son los documentos que han hasheado a dicho bucket.

Por cada tabla usamos 4 minhashes ($r = 4$) por lo tanto lo que necesitamos es una función de hashing de la forma:

$$LSH(h1, h2, h3, h4) \rightarrow [0...999999]$$

Esta función es una función de hashing tradicional y sabemos construirla a partir de una familia universal. Vamos a tener en total 6 de estas funciones, una por cada tabla de hash. Al procesar los documentos lo queharemos es aplicar cada una de estas 6 funciones que nos dará una posición dentro de la tabla en donde vamos a agregar el id del documento en cuestión, es decir que cada documento se almacena en alguna posición de cada una de las 6 tablas.

Cuando tenemos un documento "query" computamos sus 24 minhashes y luego aplicamos las 6 funciones LSH, accedemos a los buckets correspondientes a las tablas de hash y recuperamos de cada una todos los documentos que son candidatos a ser similares al documento en cuestión. Opcionalmente podemos calcular la semejanza real de nuestro query contra los documentos recuperados o simplemente devolverlos.

Desarrollamos un ejemplo numérico para explicitar como funciona la función LSH. Suponemos que usamos 8 minhashes con $r = 2$ y $b = 4$, es decir que vamos

a tener 4 tablas de hash, el tamaño de cada tabla es tan grande como podamos manejar en memoria ya que esto nos ayuda a minimizar las colisiones entre puntos que no son realmente semejantes.

Si tenemos un query q que es un texto por ejemplo "hoy es lunes" y usamos shingles de 5 caracteres entonces el primer paso es a partir del texto obtener los shingles: {"hoy e", "oy es", "y es", "es l", "es lu", "s lun", "lune", "lunes"}. Luego a cada shingle le tenemos que aplicar cada una de las 8 funciones de hashing supongamos que usamos números de 8 bits para hashear los shingles (en un caso real usaríamos 32 o 64).

shingle	h1	h2	h3	h4	h5	h6	h7	h8
hoy e	43	148	259	62	206	24	141	171
oy es	248	103	5	236	261	258	140	221
y es	181	88	136	19	138	97	5	94
es l	121	143	89	182	41	178	98	64
es lu	239	248	188	237	20	130	226	228
s lun	161	85	94	129	230	150	100	42
lune	122	119	5	37	56	91	113	261
lunes	189	134	113	18	50	56	200	23
minhash	43	85	5	18	20	24	5	23

Table 6.7: Valores de $h(\text{shingle})$ para cada shingle de q

Los 8 valores de minhash son los que nos interesan para acceder ahora a las 4 tablas de hash que estamos usando ($b = 4$). En la primer tabla vamos a acceder al bucket apuntado por $h(43,85)$, en la segunda $h(5,18)$, en la tercera $h(20,24)$ y finalmente $h(5,23)$.

$h(43,85)$	109873
$h(5,18)$	70
$h(20,24)$	88541
$h(5,23)$	908

Table 6.8: Valores de la función LSH para cada tabla

Tenemos entonces que acceder a estos buckets en cada tabla y ver que documentos se encuentran en dichos buckets:

Por lo tanto los documentos candidatos son: {d8,d203,d12,d9873,d16,d17,d167,d800,d1003,d1024} el último paso es calcular la distancia de Jaccard entre el query q y cada uno de estos documentos para finalmente ordenarlos por distancia, de esta forma podemos devolver el vecino mas cercano al query o los k vecinos mas cercanos.

Este procedimiento se va a repetir en todas las familias de funciones LSH en donde lo único que vamos a cambiar es la forma de calcular el minhash según la distancia que estemos usando.

A continuación desarrollaremos la teoría de lo que acabamos de ver en forma práctica.

reg	tabla1	tabla2	tabla3	tabla4
...
70	...	d8,d203
...
908	d12,d8,d9873
...
88541	d16,d17,d167,d800	...
...
109873	d12,d8,d1003,d1024
...

Table 6.9:

6.4 Familias de Funciones LSH

La técnica que hemos usado para la construcción de funciones LSH basándonos en minhashes para la distancia de Jaccard nos define una familia de funciones LSH:

$$(d1, d2, p1, p2)$$

Donde $d1$ y $d2$ son distancias de Jaccard. $p1$ es la probabilidad de que dos documentos sean candidatos si su distancia es igual o menor a $d1$ y $p2$ es la probabilidad de que dos documentos sean similares si su distancia es igual o mayor a $d2$.

Usando un solo minhash sabemos que la distancia de Jaccard es 1 menos la semejanza y la probabilidad de que dos documentos sean similares es igual a la semejanza por lo tanto para un solo minhash nuestra familia de funciones LSH es la siguiente:

$$(d1, d2, 1 - d1, 1 - d2)$$

Si usamos b bandas de r minhashes cada uno entonces nuestra familia es:

$$(d1, d2, 1 - (1 - (1 - d1)^r)^b, 1 - (1 - (1 - d2)^r)^b)$$

Que es simplemente aplicar la fórmula que habíamos deducido a $1 - d1$ y $1 - d2$

Así como hemos agrupado nuestras funciones en b bandas de r funciones cada una podríamos armar construcciones mas complejas. Por ejemplo podemos tener 6 grupos de 3 minhashes cada uno y esos 6 grupos dividirlos a su vez en 2 meta-grupos de 3 grupos cada uno. Tenemos en total 18 minhashes m1..m18.

Para que dos documentos sean similares podemos definir como criterio que los mismos sean similares en ambos meta-grupos. (AND) Para que dos documentos sean similares en un metagrupo definimos que sean similares en alguno de sus grupos (OR) Para que dos documentos sean similares en un grupo definimos que sus 3 minhashes coincidan (AND)

Por ejemplo si m₄,m₅ y m₆ coinciden entonces los documentos son similares en el grupo 2 y como son similares en el grupo 2 son similares en el metagrupo que agrupa los grupos 1,2 y 3. Por otro lado si m₁₀,m₁₁ y m₁₂ coinciden entonces los documentos son similares en el grupo 4 y por lo tanto son similares en el metagrupo que agrupa los grupos 4,5 y 6. Como los documentos son similares en el metagrupo 1 y el 2 los documentos son candidatos a ser similares.

Calculemos entonces la probabilidad de que dos documentos sean similares en este esquema.

La probabilidad de que los documentos sean candidatos en un meta grupo la sabemos calcular, cada metagrupo tiene 3 grupos de 3 minhashes cada uno:

$$1 - (1 - p^3)^3$$

Y como tenemos dos metagrupos y queremos que sean candidatos en ambos:

$$(1 - (1 - p^3)^3)^2$$

Veamos que resultado nos da esto para distintos valores de p

P	$(1 - (1 - p^3)^3)^2$
0.1	0.000008
0.2	0.00005
0.3	0.0062
0.4	0.032
0.5	0.1090
0.6	0.2684
0.7	0.5132
0.8	0.7811
0.9	0.9606

Table 6.10: Buscando b y r.

Como vemos en la tabla este esquema sería conveniente si queremos que sean candidatos documentos cuya semejanza es 0.8 o mayor pero que no sean semejantes los documentos que son 0.6 similares o menos.

Armando diferentes tipos de construcciones AND y OR podemos armar cualquier tipo de familias de funciones LSH para la distancia de Jaccard de acuerdo a los requisitos que tengamos en cuanto a la semejanza de los documentos para ser candidatos.

En la práctica en general los falsos negativos son mucho mas costosos que los falsos positivos. Los falsos positivos siempre los podemos eliminar calculando la semejanza entre los documentos candidatos y filtrando a los que eran candidatos pero realmente no eran similares (colisiones). Los falsos negativos son un problema mas grave ya que son documentos similares que no recuperamos y que no tenemos forma alguna de recuperarlos. Por este motivo en la práctica una solución standard usaría $r = 1$ y $b = n$ que puede ser un valor entre 5 y 20

según la cantidad de espacio que tengamos para almacenar las b tablas de hash y el nivel de recall que querramos tener al consultar por documentos similares a otro.

6.5 LSH para la distancia angular

Una métrica muy popular especialmente para textos y strings representados como vectores es distancia angular, la distancia que nos interesa aquí es el ángulo entre los vectores, el coseno entre los vectores sirve entonces como medida de semejanza, dos vectores que tienen igual dirección tienen distancia angular 0 y coseno igual a 1 mientras que vectores que apuntan en direcciones opuestas tienen distancia angular 180 grados y coseno=-1. El coseno entre dos vectores se calcula como:

$$\cos(\theta) = \frac{\dot{X}Y}{|X||Y|}$$

Para aproximar la distancia angular usando LSH vamos a ver dos métodos, uno basado en hiperplanos al azar [Charikar] y otro un tanto mas complejo que se basa en polítopos regulares [Andoni2].

6.5.1 El método de los hiperplanos

Para generar un minhash válido para la distancia coseno vamos a calcular el producto interno entre nuestro vector y un vector aleatorio que puede estar formado simplemente por valores +1 y -1 ya que estos son suficientes para definir un hiperplano aleatorio. El valor del minhash va a ser 1 si el resultado del producto interno es positivo y -1 si el resultado es negativo.

Por ejemplo si nuestro vector es $(3, -2, 3, 4)$ generamos un vector al azar tipo $(1, -1, -1, 1)$ y el producto interno entre los vectores es 9 por lo tanto el valor del minhash es +1.

Interpretemos ahora el significado del minhash. La probabilidad de que dos vectores proyectados al azar sobre un hiperplano tengan diferente signo es igual al ángulo entre ellos dividido sobre 180: $\theta/180$ ya que cuanto mas chico es el ángulo mas probable es que las proyecciones tengan igual sentido. Por lo tanto la probabilidad de que los minhashes coincidan para dos vectores es:

$$1 - \theta/180$$

Por lo tanto con un solo minhash el método de las proyecciones aleatorias nos define la siguiente familia de funciones LSH:

$$(d1, d2, 1 - d1/\pi, 1 - d2/\pi)$$

En donde $d1$ y $d2$ son las distancias angulares entre los vectores.

De la misma forma en que extendimos LSH para la distancia Jaccard podemos hacerlo para la distancia coseno. En lugar de usar 1 minhash usamos varios,

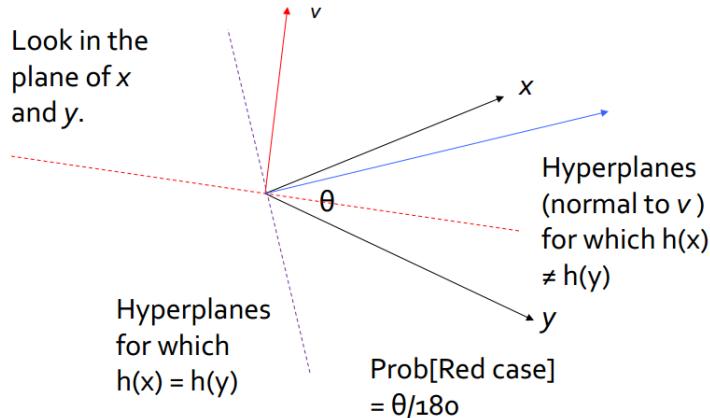


Figure 6.7: LSH para la distancia coseno

la distancia angular se puede estimar en función de la cantidad de minhash que coinciden. Por ejemplo si usamos 4 minhash tenemos 4 posibles casos:

minhashes que coinciden	ángulo
0/4	180
1/4	135
2/4	90
3/4	45
4/4	0

Table 6.11: Estimando θ usando minhash.

En realidad ni el ángulo ni el coseno realmente importan ya que lo que usaremos es la familia LSH de la misma forma que lo hacíamos con la distancia Jaccard: usando b grupos de r minhashes cada uno.

6.5.2 El Método de los Polítopos

Este método es un poco mas eficiente que el método de los hiperplanos aleatorios pero es también bastante mas complejo por lo que tiene sentido cuando tenemos vectores cuya dimensionalidad es muy grande, por ejemplo vectores dispersos en miles o cientos de miles de dimensiones.

El primer paso es aplicar "the hashing trick" para reducir la dimensionalidad de los datos de $\mathbb{R}^d \rightarrow \mathbb{R}^k$ con $k \ll d$.

Una vez que hicimos esto se normalizan los vectores de forma tal que todos tengan norma 1 y entonces estamos trabajando sobre la hiper-esfera unitaria.

Dentro de esta esfera vamos a calcular los minhashes de la siguiente forma:

1. Aplicamos a nuestro vector una rotación al azar, es decir lo multiplicamos por una matriz de $k \times k$ que tenga valores aleatorios de una distribución gaussiana.
2. El resultado es un nuevo vector en k dimensiones, el valor del minhash es el número de vértice del polítopo de k dimensiones que este mas cerca del vector.

Un polítopo está delimitado por los vectores de la base unitaria con y sin signo. En 2d la base unitaria es $\{(1,0), (0,1)\}$ por lo tanto el polítopo es $\{(0,1), (0,-1), (1,0), (-1,0)\}$ es decir un cuadrado rotado. Numerando los vértices por ejemplo 0,1,2,3 si el resultado de aplicar una rotación al azar a nuestro vector fuera $(0.25, -12)$ entonces el vértice mas cercano del polítopo en 2d es $(0, -1)$ y el valor del minhash es por lo tanto 1 (el vértice cuyo número de orden es 1). Notemos que en d dimensiones el polítopo tiene siempre $2d$ vértices. En 3d el polítopo es un octahedro que tiene 6 vértices y 8 caras y que es el dual del cubo que tiene 8 vértices y 6 caras.

Ejemplo: Sea el siguiente punto en 2d $x = (12, -7)$, generamos una matriz de rotación al azar de 2d:

$$R_1 = \begin{pmatrix} 0.4572 & 0.3003 \\ -0.0146 & -0.3581 \end{pmatrix}$$

Calculamos $R_1 * x = (-0.0612, -4.2223)$

Si los vértices son: $\{0:(0,1); 1:(0,-1); 2:(1,0); 3:(-1,0)\}$ entonces el mas cercano es $(0, -1)$ y el valor del minhash es 1. Podemos generar tantos minhashes como necesitemos generando mas rotaciones al azar. Luego podemos amplificar con los métodos que ya hemos visto agrupando minhashes.

Para hacer mas eficiente este método en lugar de generar una matriz aleatoria de $k \times k$ para realizar la rotación los autores proponen usar la matriz: $H * D1 * H * D2 * H * D3$ en donde H es la transformación de Hadamard y D_i es una matriz diagonal con $+/-1$ al azar. La transformación de Hadamard se puede calcular en $O(k \log k)$ y esto es mas rápido que el producto por una matriz aleatoria de $k \times k$ que es $O(k^2)$. Empíricamente se observa que el resultado de aplicar $H * D1 * H * D2 * H * D3$ a un punto es equivalente a realizar una rotación aleatoria.

Otra variante sugerida en [Kennedy] es aplicar en primer lugar una reducción de dimensiones usando Johnson-Lindenstrauss para llevar los datos de d a m dimensiones ($m \ll d$). Y luego aplicar una rotación que al mismo tiempo aumente la dimensionalidad de los datos mediante una matriz aleatoria de $m \times d'$. Siendo d' la dimensionalidad sobre la cual comparamos contra el polítopo mas cercano.

En definitiva partiendo de un set de d dimensiones, si queremos un minhash en el rango de d' dimensiones que sea un vértice del polítopo de d' dimensiones mas cercano al punto tenemos como opción aplicar JL, The Hashing Trick o

una rotación. El problema de las rotaciones aleatorias es que son costosas ya que implican el producto de un vector con una matriz de rotación que es densa, por esto puede resultar conveniente reducir la dimensionalidad del set de datos a una dimensión m con $m < d' < d$, para hacer esta reducción podemos usar una proyección de JL o bien directamente the hashing trick. Luego la rotación la podemos hacer mediante una matriz aleatoria o nuevamente aplicando the hashing trick ya que, aunque no lo hemos demostrado, podemos usar the hashing trick para aproximar una rotación al azar si los puntos están sobre la esfera unitaria. Esto nos da muchas combinaciones posibles como por ejemplo:

1. Aplicar the hashing trick para pasar de d a d' dimensiones.
2. Reducir primero a m dimensiones via JL y luego aplicar una rotación de mxm
3. Aplicar the hashing trick para pasar de d a d' dimensiones y luego una rotación al azar mediante una matriz de $d'xd'$
4. Aplicar JL para pasar a m dimensiones y luego aplicar the hashing trick para pasar a d' dimensiones
5. Aplicar the hashing trick para pasar de d a m dimensiones y luego una rotación al azar de mxm

En todos los casos el paso final una vez que llegamos a d' dimensiones es determinar el vértice del polítopo de d' dimensiones mas cercano a nuestro punto como valor del minhash.

Todas estas combinaciones pueden funcionar bien en la práctica, para evaluar la mas conveniente debe analizarse por un lado la performance del método elegido y por otro la probabilidad de colisión en función de la distancia a la cual estaban los puntos originales. Queremos que puntos cercanos tengan una probabilidad de colisionar alta y puntos lejanos una probabilidad cercana a cero. Cuando hablamos de colisionar en este método estamos hablando de que ambos puntos tengan al mismo vértice del polítopo como vecino mas cercano.

6.6 LSH para la distancia euclideana

La distancia euclideana es sin dudas una de las métricas mas usadas, en esta sección vamos a explicar mecanismos para aproximar la distancia euclideana usando LSH, algunos de estos mecanismos también sirven para la distancia Manhattan o norma L1.

Vamos a ver un método similar al que vimos para la distancia angular basado en proyecciones aleatorias y luego otro método basado en el algoritmo de clustering K-Means.

6.6.1 El Método de las Proyecciones Aleatorias

La idea es similar a la usada para la distancia angular, vamos a usar proyecciones aleatorias, en este caso los hiperplanos aleatorios surgirán de una distribución estable, una distribución normal funciona perfectamente. Cada vector entonces lo vamos a multiplicar por un hiperplano aleatorio para obtener un número. Y el número obtenido lo vamos a discretizar a un número de bucket dividiendo por un cierto parámetro w . El resultado es el número de bucket o minhash para la distancia euclideana.

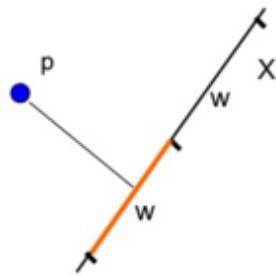


Figure 6.8: LSH para la distancia euclideana

La fórmula para calcular el minhash es entonces:

$$mh_i(x) = \left\lfloor \frac{x * v_i + a}{w} \right\rfloor$$

Donde w es un hiper-parámetro que depende de los datos, a es un número entre 0 y $w - 1$ y v_i es un vector aleatorio de igual dimensionalidad que nuestros puntos.

Cada proyección aleatoria implica un particionamiento del espacio en el cual estamos trabajando, diferentes proyecciones generan diferentes particiones:

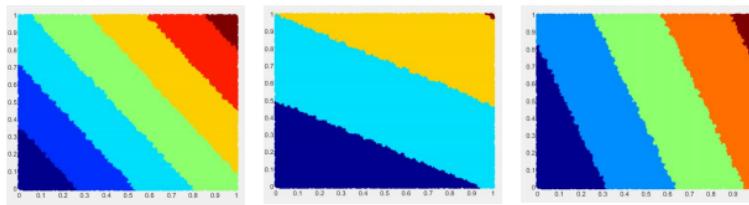


Figure 6.9: Diferentes proyecciones aleatorias

En cada proyección los puntos que están en una misma franja son considerados similares. El parámetro w regula el ancho de las bandas y la forma de discretizar y es dependiente de los datos, es decir que tenemos que calcularlo para nuestro set de datos específicamente.

Analizaremos ahora la probabilidad de que dos documentos sean similares de acuerdo a la forma en que hemos definido los minhashes para la distancia euclídea.

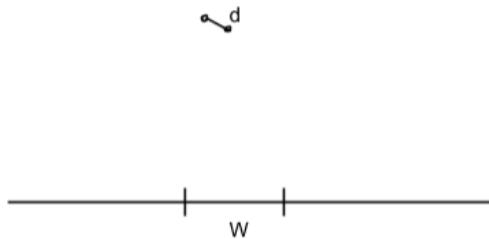


Figure 6.10: Minhash para la distancia euclideana

Cuando la distancia entre dos puntos es menor a $w/2$ la probabilidad de que caigan en el mismo bucket es $1/2$. Esto se puede demostrar fácilmente de forma geométrica ya que si los puntos están a distancia $w/2$ o inferior la única forma de que caigan en diferentes buckets es que uno de los puntos caiga fuera del bucket y el otro dentro del mismo y como el segmento nunca mide más de $w/2$ en promedio la probabilidad de que esto ocurra nunca puede ser mayor a $1/2$.

Cuando la distancia es mayor a $2w$ los puntos solo caen dentro del bucket si su ángulo es cercano a 90 grados. En concreto si el ángulo es menor a 60 grados los puntos nunca pueden estar dentro del mismo bucket y por lo tanto podemos decir que la probabilidad es $1/3$ ya que esto ocurre en $1/3$ de los ángulos posibles.

Por lo tanto nuestra familia LSH es de la forma:

$$LSH(w/2, 2w, p \geq 1/2, p \leq 1/3)$$

Pese a que esta familia tiene una definición no muy precisa igual podemos aplicar la misma amplificación que vimos en las secciones anteriores usando varios minhashes agrupándolos de la forma que nos resulte más conveniente. Cuando usamos $r = 1$ y $b = n$ que es el caso más popular lo que estamos haciendo es devolviendo como similares a los documentos que están en la misma franja en alguna de las n proyecciones aleatorias que usamos.

6.6.2 Métodos basados en K-Means: KLSH

K-Means es un algoritmo de clustering que veremos más adelante, para un conjunto de m puntos en d dimensiones y una cierta cantidad de clusters k que el usuario debe especificar K-Means genera k clusters y devuelve cuáles son los

k centroides correspondientes a cada cluster y a que centroide queda asignado cada uno de los m puntos. El algoritmo lo vamos a desarrollar cuando veamos clustering, lo tomaremos aquí como un modelo de caja negra que nos genera k centroides.

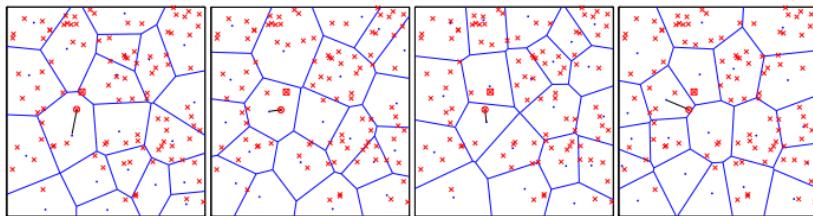


Figure 6.11: Ejemplo KLSH. El punto query es q y nos interesa encontrar su vecino mas cercano. Podemos ver que en la segunda versión de K-Means el query queda dentro del mismo cluster que su vecino mas cercano por lo que accediendo al mismo vamos a poder encontrarlo, en los otros casos encontramos otros puntos pero no el mas cercano. Este es un ejemplo de usar $r = 1$ y $b = 4$ lo cual nos da muy buena precisión pero notar que cualquier punto dentro del mismo cluster que el query debe ser comparado contra el mismo para poder encontrar el verdadero vecino mas cerano.

El centroide al cual queda asignado cada punto es un número entre 1 y k y nos sirve como minhash. Para generar varios minhashes podemos usar K-Means n veces con diferentes inicializaciones, como el algoritmo converge a un mínimo local que es dependiente de la inicialización tendremos valores diferentes en cada caso.

Con estos minhashes podemos proceder exactamente como lo hacíamos antes, generando b tablas de hash y usando r minhashes por cada tabla en donde el valor de cada minhash es el número de cluster (centroide) al cual fue asignado el punto.

Cuando queremos buscar los vecinos mas cercanos a un query q simplemente buscamos el centroide mas cercano de cada corrida de K-Means y eso nos da el minhash correspondiente al query. Luego procedemos como siempre.

6.6.3 Métodos basados en K-Means: K-Means jerárquico

Es posible aplicar K-Means de forma jerárquica a nuestro set de datos, por ejemplo a partir de m puntos corremos K-Means con $k = \sqrt{m}$ centroides, esto nos da, teóricamente, un promedio de \sqrt{m} puntos por centroide. Para los puntos asignados a cada centroide podemos repetir el proceso aplicando K-Means nuevamente, esto nos permite crear un árbol de K-Means. Cuando queremos resolver los vecinos mas cercanos a un query q empezamos por encontrar el centroide mas cercano en el primer nivel y luego bajamos a los puntos asignados a dicho centroide en donde buscamos el centroide mas cercano nuevamente y

así recursivamente. Este método es eficiente para casos en los cuales tenemos billones de puntos. Para mejorar la precisión del método podemos usar multi-probe que desarrollaremos mas abajo.

Este método lo vamos a volver a desarrollar al ver K-Means.

6.7 LSH para la distancia Hamming

La distancia de Hamming indica simplemente el número de dimensiones en las cuales dos vectores difieren. Es igual a la norma l_0 . Por ejemplo la distancia de Hamming entre $(2, 3, 4, 1)$ y $(2, 5, 0, 1)$ es 2.

Para la distancia de Hamming existe una forma inmediata de obtener un minhash, simplemente seleccionamos al azar alguna de las dimensiones del vector. La probabilidad de que los minhashes no coincidan es igual a la distancia de Hamming entre los vectores dividido por su longitud. Por lo tanto la probabilidad de que dos minhashes coincidan es:

$$p = 1 - \frac{H(x, y)}{\text{len}(x)}$$

Esto nos define la siguiente familia para vectores de k dimensiones.

$$\text{LSH}(d1, d2, 1 - d1/k, 1 - d2/k)$$

Podemos ahora simplemente amplificar esta familia usando la técnica que ya hemos visto combinando ORs y ANDs de los diferentes minhashes.

6.8 Multi-Probe LSH

Hasta el momento todas las funciones LSH que hemos visto se basan en la construcción de minhashes y luego en la búsqueda de vecinos en una cantidad de tablas de hash b en donde por cada tabla accedemos a un número de bucket hasheando r minhashes, esto equivale a decir que un punto es candidato a ser vecino de otro si en alguna de las b tablas sus r minhashes coinciden.

Para mayor precisión es decir que no haya falsos negativos es necesario usar un valor de b grande, en general en el orden de entre 100 o 1000, pero esto queda limitado por el tamaño de la memoria, el método de multi-probe intenta aumentar la precisión de una función LSH sin necesitar una cantidad de tablas de hash muy grande.

La idea es no acceder a un solo bucket por tabla sino a varios, siendo la probabilidad de acceder a un bucket proporcional a la distancia entre el query y los puntos del bucket, distancia que por supuesto desconocemos.

El objetivo es entonces dado un query q determinar cuáles son los buckets en donde hay mayor probabilidad de encontrar vecinos cercanos y acceder a una cierta cantidad T de estos buckets. El bucket mas probable sabemos encontrarlo simplemente aplicando una función de hashing a los r minhashes de q , el

problema es como determinar cuáles son los mejores buckets a probar además de éste.

La respuesta depende, por supuesto, de la métrica empleada, a continuación veremos como podemos hacer multi-probing según el tipo de minhash generado.

6.8.1 Multi-Probing para la distancia euclideana, método de proyecciones aleatorias

En este caso el minhash surge de la división por w , por lo tanto los valores que nos interesan son los vecinos al resultado del minhash, es decir que si el minhash es (4) nos interesa el bucket 4 y luego el 3 y 5, luego el 2 y el 6, etc. Si usamos r minhashes podemos entonces generar buckets candidatos modificando $+/- 1$ el valor de cada minhash. Por ejemplo si tenemos (3,12,9) podemos probar los buckets (3,12,9), (2,12,9), (4,12,9), (3,13,9), (3,11,9), (3,12,8), (3,12,10), etc.

6.8.2 Multi-Probing para la distancia euclideana, métodos basados en K-Means

En este caso el esquema es muy simple ya que calculamos la distancia entre el query y cada centroide para saber el valor del minhash que es el centroide que está a menor distancia. Por lo tanto los minhashes alternativos son los números de centroides que siguen ordenados por distancia decreciente.

En todos los métodos basados en K-Means podemos implementar Multi-Probing de forma sencilla accediendo a T centroides en lugar de uno solo.

6.8.3 Multi-Probing para la distancia angular, método de proyecciones aleatorias

En este caso el minhash solo puede tomar dos valores: +1 o -1, supongamos que tenemos el caso (+1,-1,+1) usando 3 minhashes por tabla, si queremos aplicar multi-probing lo que podemos hacer es modificar alguno de los minhashes cambiando el signo y acceder a dicho bucket.

6.8.4 Multi-Probing para la distancia angular, método de los polítopos

En el método de los polítopos podemos aplicar multi-probing accediendo a los T vértices mas cercanos en lugar del mas cercano, este es un procedimiento sencillo y que no cuesta nada ya que para determinar el vértice mas cercano es necesario calcular la distancia entre cada vértice y la rotación aleatoria del punto original, por lo tanto simplemente en lugar de un vértice usamos T .

6.8.5 Multi-Probing para la distancia de Jaccard

En la distancia Jaccard la cuestión no es simple, una posible variante es realizar alguna perturbación en el string (texto) query y ver cuales serían los minhashes

para el mismo. Por ejemplo reemplazando alguna palabra por su sinónimo, cambiando de singular a plural etc. Cada perturbación nos da un string "similar" al que estamos buscando y por eso tiene sentido acceder a los buckets apuntados por dichos strings para buscar textos similares al que nos interesa. El método de perturbar el dato original para generar otros minhashes es genérico y podría usarse en cualquier esquema de LSH si fuera necesario.^{2s}

Multi-Probing nos permite ganar precisión sacrificando tiempo. Al buscar en mayor cantidad de buckets disminuye la probabilidad de falsos negativos pero agregamos tiempo al tener que acceder a mas buckets, recuperar mas puntos y tener que calcular las distancias entre el query y estos puntos. La técnica es muy interesante porque permite ganar precisión sin sacrificar memoria.

6.9 LSH Conclusiones

Es hora de cerrar esta parte del capítulo haciendo un resumen de lo visto hasta ahora.

LSH tiene como objetivo resolver el problema de los k vecinos mas cercanos de forma eficiente, idealmente en $O(1)$ o similar usando la menor cantidad de espacio posible, en muchos casos esto implica una aproximación al problema ya que la solución que devolvemos puede no ser exacta. Esta inexactitud puede deberse a la presencia de puntos superfluos (falsos positivos) o la ausencia de puntos relevantes (falsos negativos). Un falso-positivo es un punto que no es cercano al que estamos buscando pero que LSH devuelve como similar. Los falsos-positivos se filtran facilmente simplemente calculando la distancia real entre el punto query q y los puntos candidatos a ser similares devueltos por LSH, estas distancias siempre las tenemos que calcular para saber cuales son los k vecinos mas cercanos. El problema de los falsos-positivos es entonces de performance, si tenemos muchos falsos positivos tenderemos muchos puntos irrelevantes cuya distancia contra el query tenemos que calcular.

Un falso-negativo es un punto que es cercano al query pero que LSH no puede encontrar. Esto afecta la *precisión* del método. Los falsos-negativos no tienen solución ya que la única forma de buscarlos sería recorrer todos los datos y si hacemos eso entonces estamos en KNN por fuerza bruta y no necesitamos LSH.

En general todas las funciones LSH que definimos se caracterizan por tener una forma de calcular un minhash a partir del dato y luego definir los siguientes parámetros:

- b que es la cantidad de tablas de hash a usar
- r que es la cantidad de minhash a usar por tabla
- T que es la cantidad de buckets a probar en cada tabla (multi-probing)

Si queremos disminuir los falsos negativos, es decir ganar precisión, podemos aumentar b sacrificando memoria o aumentar T sacrificando velocidad.

En general vamos a definir b en función de la memoria que tengamos disponible, cuanto mayor es b mayor precisión va a tener el algoritmo. Luego una vez fijado b tenemos que definir r y T en función de la cantidad de falsos positivos y falsos negativos que podamos manejar y la performance (velocidad) de las consultas.

Para un b fijo aumentar r disminuye los falsos positivos y también disminuye la precisión. Aumentar T mejora la precisión pero disminuye la performance.

6.10 LSH Para Máxima semejanza

En líneas generales LSH es muy eficiente para recuperar elementos similares cuando usamos distancias relativamente amplias pero si queremos encontrar los elementos iguales o casi iguales a otros es decir con una semejanza muy grande puede resultar ineficiente en comparación con otros algoritmos. En esta sección exploraremos algunas alternativas usando como base la distancia de Jaccard entre documentos.

Dado un documento podemos extraer del mismo un conjunto de shingles que pertenecen al conjunto de shingles totales de nuestra colección. Si ordenamos todos los shingles en la colección podemos asociar a cada shingle un número, por ejemplo "HIS T" = 1094. Podemos representar entonces un documento como un conjunto de números que son los shingles que representan al documento ordenándolos de menor a mayor. En general el mejor orden para los shingles es por frecuencia descendente, de esta forma el shingle 23 es mucho mas raro que el shingle 193283. Para simplificar la explicación que sigue supongamos que solo existen 26 shingles posibles entonces podemos asociar cada uno con una letra del abecedario y cada documento nos queda representado por un string. Por ejemplo "acfj" es un documento que tiene el shingle 0, el 2, el 5 y el 9. Podemos entonces encarar la semejanza entre documentos como si fuera un problema de semejanza entre strings.

6.10.1 Usando la Longitud

Supongamos que queremos recuperar documentos que tengan semejanza de Jaccard 0.9 o superior con respecto a un cierto documento query. Según la longitud del documento query es decir la cantidad de shingles que tiene sabemos que longitudes pueden tener los documentos que pueden ser similares. Por ejemplo si nuestro documento tiene longitud $L_s = 10$ entonces un documento con $L_s = 9, L_s = 10$ o con $L_s = 11$ shingles puede ser similar pero un documento con 8 o 12 shingles no ya que a lo sumo tendremos $8/10$ o $10/12$ shingles en común y $8/10 < 0.9$ y $10/12 < 0.9$

Como primera solución podemos entonces indexar cada documento en un bucket de acuerdo a su longitud y luego comparar el documento query únicamente contra aquellos documentos que están en los buckets cuya longitud puede satisfacer la consulta. Estos documentos pueden ser muchos, incluso millones.

6.10.2 Usando Prefijos

Sigamos pensando que queremos documentos cuya semejanza con un cierto documento query es 0.9 o mayor llamemos a este número J ya que es la semejanza de Jaccard. Supongamos que nuestro documento tiene 10 shingles. Queremos mostrar que para que un documento sea similar éste debe tener alguno de los dos primeros shingles. Si no tuviera ninguno de los dos primeros shingles entonces solo podría tener 8 shingles en común y la semejanza de Jaccard sería a lo sumo $8/10$ que es menor a J . Por lo tanto si $J = 0.9$ y nuestro texto tiene 10 shingles alcanzaría con indexar los primeros 2 shingles del mismo. Con textos mayores es necesario indexar mayor cantidad de shingles, consideremos un texto que tiene 50 shingles, para que la semejanza sea mayor o igual a $J = 0.9$ contra algún documento query desconocido necesitamos que al menos 45 de los 50 shingles estén en la intersección por lo tanto necesitamos indexar los primeros 6 símbolos del documento. En concreto la cantidad de símbolos a indexar es:

$$p = \lfloor (1 - J)L_s \rfloor + 1$$

Donde recordemos L_s es la longitud del texto en shingles.

Con este método vamos entonces a crear un bucket por cada shingle que haya que indexar y luego indexamos a cada documento p veces poniendo el id del mismo en los buckets que corresponden a sus primeros p shingles. Para hacer una consulta dado un cierto documento calculamos p y luego buscamos en los buckets correspondientes a los primeros p shingles del documento query.

6.10.3 Prefijos y Posiciones

Consideremos los strings: $s = acdefghijk$ y $t = bcdefghijk$, usando $J = 0.9$ como tienen longitud 10 ambos serían indexados por sus dos primeros shingles (representados por a,c y b,c respectivamente). Por lo tanto como coincidiríamos en el bucket de "c" los documentos serían candidatos a similares. Sin embargo la semejanza de Jaccard entre s y t es $9/11 = 0.81$ que es menor que $J = 0.9$

Supongamos que construimos nuestro índice en base al par ordenado (Shingle, posición) es decir que para s usaríamos $(a,0)$ y $(c,1)$ mientras que para t usaríamos $(b,0)$ y $(c,1)$. Analicemos ahora como se haría una consulta. Empecemos con nuestro string $s = acdefghijk$, queremos encontrar strings que sean similares en $J = 0.9$ o mas. Sabemos que como $L_s = 10$ entonces $p = 2$ por lo que solo consideraremos los dos primeros caracteres (shingles) en nuestro string: a y c. Tenemos potencialmente que buscar en 4 buckets: $(a,1)$ $(a,2)$ $(c,1)$ y $(c,2)$ para encontrar documentos candidatos pero en realidad de esas 4 combinaciones hay una que no es posible que es $(c,2)$. Si ya evaluamos $(a,1)(a,2)$ y $(c,1)$ sin encontrar candidatos entonces sabemos que no hay documentos que comiencen con el mismo shingle "a" que s , ya que buscamos $(a,1)$ sin éxito. También sabemos que no hay documentos que comiencen con el segundo shingle de s que es "c" ya que buscamos $(c,1)$ sin éxito y sabemos que no hay documentos que tengan el primer shingle de s en la segunda posición pues hemos buscado $(a,2)$ por lo tanto un documento que tenga a "c" en la segunda posición no puede

tener "a" en la primera es decir que vamos a tener al menos 11 shingles en la unión y de estos 11 shingles solo pueden coincidir 9 por lo tanto el documento no puede ser similar con $J = 0.9$ un ejemplo es: $x=bcdefghijk$.

Desarrollemos esto de forma teórica sea L_s la longitud del documento (cantidad de shingles) J la semejanza de Jaccard que buscamos como mínimo, i la posición del shingle en el documento query. Queremos hallar j que es la posición máxima que puede tener el shingle i en un documento candidato a ser similar.

Para calcular la cota superior para j supongamos que t es un documento cuyos primeros $j - 1$ símbolos no matchearon nada en s pero el i -ésimo símbolo de s es igual al j -ésimo símbolo de t la semejanza máxima entre s y t se da cuando todos los símbolos a partir de i y j respectivamente son idénticos. Por ejemplo si $s=ad^*$ y $t=bfd^*$ entonces $i = 2$, $j = 3$ y para que la semejanza sea máxima lo que sigue luego de d en ambos strings debería coincidir. El tamaño de la intersección es entonces: $L_s - i + 1$ y la unión es $L_s + j - 1$ Esto es porque s contribuye su longitud L_s a la unión y además hay $j - 1$ símbolos que no están en s por lo tanto la semejanza máxima de Jaccard es:

$$J \leq \frac{L_s - i + 1}{L_s + j - 1}$$

Despejando j tenemos:

$$j \leq (L_s(1 - J) - i + 1 + J)/J$$

Reemplazemos para verificar que funciona bien en nuestro ejemplo con $s=acdefghijk$. $L_s = 10$, $J = 0.9$

Si $i = 1$ entonces $j \leq (10(0.1) - 1 + 1 + 0.9)/0.9$ entonces $j \leq 2.11$ por lo que tenemos que analizar (a,1) y (a,2). Si $i = 2$ entonces $j \leq (10(0.1) - 2 + 1 + 0.9)/0.9$ entonces $j \leq 1$ por lo que tenemos que analizar (c,1) únicamente.

6.10.4 Prefijos, Posiciones y Longitud

En la sección anterior consideramos el caso ideal en el cual una vez que encontramos un match entre un shingle de un string y otro todo lo que sigue es igual, pero evidentemente no siempre es este el caso y sería deseable evitar estos falsos positivos. Una forma de hacer esto es indexar no solo el símbolo y su posición sino también la cantidad de shingles que siguen a continuación.

Por ejemplo si $s = acdefghijk$ entonces indexaríamos (a,1,9) y (c,2,8).

Sea un string query s de longitud L_s y sea J la semejanza de Jaccard que buscamos. Sea i la posición de s en la cual estamos y sea p la cantidad de shingles que quedan luego de la posición i . Queremos buscar el símbolo que estamos procesando de s en nuestros buckets y necesitamos una cota superior para j que es la posición del símbolo en el string y q que es la cantidad de símbolos que siguen luego de la posición j .

Primer caso: $p \geq q$

La cantidad máxima de shingles en la intersección es:

$$L_s - i + 1 - (p - q)$$

Ya que $L_s = i + p$ la intersección es entonces $q + 1$. La unión como mínimo es $L_s + j - 1$ igual que antes por lo tanto:

$$J \leq \frac{q + 1}{L_s + j - 1}$$

Segundo caso: $p < q$

Ahora la intersección tiene como máximo $L_s - i + 1$ tal y como calculamos cuando no considerábamos los shingles siguientes. Pero la unión ahora cambia y es $K_s + j - 1 + q - p$. Si usamos de nuevo $L_s = i + p$ podemos reemplazar $L_s - p$ por i y nos queda $i + j - 1 + q$ por lo tanto:

$$J \leq \frac{L_s - i + 1}{i + j - 1 + q}$$

Ejemplo: Sea $s=acdefghijk$ y usemos $J = 0.8$. Sabemos que $L_s = 10$ calculamos la cantidad de shingles a indexar que es igual a 3 por lo que debemos considerar $i = 1, 2, 3$ es decir los shingles a,c,d. Supongamos primero que $p \geq q$

Sea $i = 1$ entonces $p = 9$ por lo tanto $q = 9$
 si $q = 9$ entonces $10/(9 + j) \geq 0.8$ entonces $j=1,2,3$. si $q = 8$ entonces $9/(9 + j) \geq 0.8$ entonces $j=1,2$ si $q = 7$ entonces $8/(9 + j) \geq 0.9$ entonces $j=1$ si $q = 6$ no hay valor de j

Por lo tanto aquí deberíamos buscar en los buckets (a,1,9) (a,2,9) (a,3,9) (a,1,8) (a,2,8) (a,1,7)

Sea $i = 2$ entonces $p = 9$ por lo tanto $q \geq 8$. Dado $(q + 1)/(9 + j) \geq 0.8$ podemos usar el mismo análisis pero excluyendo $q = 9$ entonces:

si $q = 8$ entonces $j = 1, 2$ si $q = 7$ entonces $j = 1$

Y buscamos (c,1,8) (c,2,8) (c,1,7)

Si $i = 3$ entonces $p = 7$ y la única opción es $q = 7$ y $j = 1$

Agregamos (d,1,7)

Ahora consideremos $p < q$

si $i = 1$ entonces $p = 9$ y $q \geq 10$ con $10/(q + j) \geq 0.8$ los posibles valores de q y j son:

$q = 10$ y $j = 1, 2$ $q = 11$ y $j = 1$

Agregamos entonces: (a,1,10) (a,2,10) (a,1,11)

Si $i = 2$ la única opción es $q = 9$ y $j = 1$ que ya hemos investigado

Si $i = 3$ entonces no hay soluciones.

La figura 6.12 muestra el total de buckets a buscar que son 14.

Con este método la cantidad de falsos positivos encontrados en cada bucket disminuye sensiblemente, el contrapunto es la cantidad de espacio necesario ya que usamos una cantidad de buckets mucho mayor al usar 3 dimensiones para indexar los mismos.

Notemos que el método descripto en todas sus variantes puede tener falsos positivos pero nunca falsos negativos lo cual es una condición muy deseable.

	q	$j = 1$	$j = 2$	$j = 3$
$i = 1$	7	x		
	8	x	x	
	9	x	x	x
	10	x	x	
	11	x		
$i = 2$	7	x		
	8	x	x	
	9	x		
$i = 3$	7	x		

Figure 6.12: LSH para distancias muy pequeñas

6.11 LSH forests

Presentamos ahora una idea reciente [Bawa] en la cual usamos LSH para construir un árbol binario de búsqueda. La idea consiste en convertir cada minhash en un 0 o 1, por ejemplo según sea par o impar e ir insertando los datos en un árbol binario de búsqueda, para cada dato generamos tantos minhashes como sean necesarios para que la concatenación de todos sus minhashes sea una cadena única es decir que le corresponda una hoja del árbol. Eventualmente puede ser necesario generar muchísimos minhashes por lo que es conveniente fijar una longitud máxima para la cantidad de minhashes que puede tener un dato, en este caso puede ocurrir que una hoja del árbol tenga mas de un dato pero en general serán muy pocos.

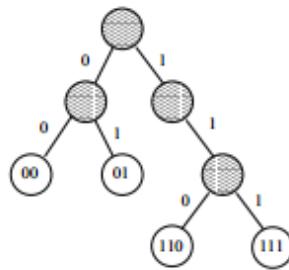


Figure 6.13: LSH Forests

Al igual que en LSH tradicional es conveniente crear no uno sino varios árboles por lo que hablamos de una estructura de tipo "bosque" (forest) y de

alli el nombre LSH-Forests.

A la hora de buscar resultados en base a un cierto query vamos aplicando los minhashes al dato y navegando en paralelo todos los árboles de nuestro forest hasta llegar en cada uno a la hoja que comparte el prefijo de mayor longitud con nuestro dato. Una vez que llegamos hasta la hoja correspondiente en cada forest podemos recuperar el o los datos que se encuentren en dicha hoja y luego hacer backtracking hacia arriba en cada árbol para ir recuperando otros datos hasta tener una cantidad suficiente de acuerdo a lo que estemos haciendo.

Esta es una estructura bastante eficiente en el cual organizamos cada tabla de hash como un árbol binario de búsqueda, como contrapunto necesitamos generar una cantidad de minhashes muy superior a los que necesitamos en LSH tradicional.

Chapter 7

Patrones Frecuentes

If we have data, let's look at
data. If all we have are opinions,
let's go with mine.

Jim Barksdale.

Este capítulo comienza con el problema de encontrar patrones que se repiten con una cierta frecuencia en una base de datos. Por ejemplo si cada "registro" es una lista de artículos adquiridos en un supermercado puede interesarnos descubrir artículos que frecuentemente son comprados juntos. El ejemplo mas clásico es el de la cerveza y los pañales que ya hemos mencionado en el capítulo 1. A este tipo de operaciones se las suele denominar "pattern mining" y son, en gran medida, la columna vertebral de lo que llamamos "Data Mining", que es la ciencia que se dedica a descubrir patrones a partir de un conjunto de datos.

Encontrar conjuntos de ítems que frecuentemente están juntos implica también encontrar "reglas de asociación" de tipo "quienes compran A también suelen comprar B", por ejemplo quienes compran carbón y leña también suelen comprar carne. Una regla de asociación puede escribirse de la forma $A \rightarrow B$ en donde en ambos lados podemos tener uno o varios ítems. Un conjunto de ítems frecuentes los vamos a denotar como un conjunto: $\{A, B, \dots\}$

Consideremos el siguiente ejemplo en donde listamos ítems comprados en un supermercado, cada fila es un registro y reúne las compras de un cliente.

1. pepsi, carne, tomate
2. coca, carne, papas, pollo
3. pepsi, carne, papas
4. pepsi, carne, papas, huevos
5. coca, papas, pollo
6. coca, carne, papas, tomate, lechuga
7. coca, papa, pollo
8. coca, carne, papas, huevos

El primer paso para encontrar conjuntos de ítems frecuentes (patrones) es definir un nivel de *soporte* que es la cantidad de veces mínima que el ítem ha sido comprado. Por ejemplo si definimos soporte=3 los siguientes conjuntos son frecuentes:

1. {pepsi}
2. {carne}
3. {coca}
4. {papas}
5. {pollo}
6. {pepsi,carne}
7. {carne,coca}
8. {carne,papas}
9. {coca,papas}
10. {coca,pollo}
11. {papas,pollo}

En general el soporte no es un número fijo sino un porcentaje sobre el total de registros, por ejemplo un 1% del total.

El objetivo de los algoritmos que vamos a estudiar en este capítulo es encontrar estos conjuntos a partir de los datos.

7.1 A-Priori

A-Priori es el algoritmo más popular para la generación de ítems frecuentes y para encontrar patrones de asociación. El algoritmo hace múltiples pasadas sobre los datos, en la pasada i genera tuplas de ítems frecuentes de tamaño i . Hay muchísimas versiones de este algoritmo ya que puede programarse de diferentes formas. En este capítulo veremos una de estas variantes y luego pasaremos a otros algoritmos.

En su primer pasada sobre los datos A-Priori va a contar cuantas veces aparece cada ítem individual, esto puede hacerse mediante una tabla de hash o diccionario. En nuestro ejemplo la primer pasada de A-Priori genera lo siguiente:

```
pepsi 3
carne 6
tomate 2
coca 5
papas 7
```

```
pollo 3
huevos 2
lechuga 1
```

Luego de esta primer pasada ya tenemos los ítems individuales (tuplas de tamaño 1) que son frecuentes, simplemente filtrando el resultado anterior según el soporte que necesitemos. Estos resultados se almacenan en memoria a medida que el algoritmo procesa los datos.

En la segunda pasada A-Priori va a leer de nuevo los datos y por cada registro va a generar las combinaciones de 2 elementos. Por ejemplo en nuestro primer registro {pepsi,carne,tomate} vamos a generar: {pepsi,carne}; {pepsi,tomate} y {carne,tomate}. Esto se hace simplemente con dos ciclos for anidados. La cantidad total de combinaciones es $\binom{n}{2}$ siendo n la cantidad de ítems en el registro. En general esto es siempre una cantidad aceptable. Por cada tupla candidata sabemos que solo puede ser frecuente si sus dos elementos son frecuentes. Por ejemplo {carne,tomate} no puede tener soporte 3 porque tomate tiene soporte 2, es decir que si tomate no es frecuente entonces ninguna tupla que tenga tomate es frecuente. A-Priori revisa la lista de ítems individuales frecuentes que tenemos en memoria y descarta las tuplas que no pueden ser frecuentes. Para las tuplas que sí pueden ser frecuentes creamos una segunda tabla de hash o diccionario en donde vamos contabilizando la cantidad de veces que aparecen las tuplas. El resultado de la segunda pasada de A-Priori es entonces el siguiente:

```
pepsi,carne 3
pepsi,papas 2
pepsi, pollo 2
carne, coca 3
carne, papas 5
coca, papas 5
coca, pollo 3
papas, pollo 3
```

Esta lista también debe guardarse en memoria y en general indica la cantidad de memoria máxima que puede necesitar el algoritmo. Filtrando esta lista obtenemos todas las tuplas de 2 ítems que son frecuentes.

En una tercer pasada A-Priori vuelve a procesar los datos pero esta vez generando triplas de ítems candidatos, por ejemplo para nuestro primer registro la única tripla candidata es {pepsi,carne,tomate} ya que el registro solo tiene 3 elementos. Para el registro 6 que tiene 5 elementos tenemos un total de 20 triplas candidatas $\binom{6}{3}$

Por cada tripla generada A-Priori debe chequear si la misma puede ser frecuente. Para que {A,B,C} pueda ser frecuente {A,B} {A,C} y {B,C} tienen que ser frecuentes. Esto lo podemos chequear contra la lista de duplas frecuentes que tenemos en memoria del paso anterior. Si la tripla es candidata entonces creamos un nuevo diccionario y vamos contando cuantas veces aparece cada tripla.

El resultado es:

```
carne,coca,papas 3
coca,papas,pollo 3
```

Como solo tenemos 2 triplas frecuentes sabemos que no puede haber tuplas frecuentes de cuatro elementos, ya que si $\{A,B,C,D\}$ fuera frecuente entonces tendríamos $\{A,B,C\}; \{A,C,D\}, \{A,B,D\}$ y $\{B,C,D\}$ como frecuentes. Por lo tanto en nuestro ejemplo A-Priori hace tres pasadas sobre los datos y devuelve las tuplas de 1,2 o 3 elementos que tienen el soporte pedido (3).

7.1.1 Generación de Candidatos Intermedia

En la versión de A-Priori que hemos visto la generación de candidatos se hace a nivel de cada registro y cada tupla candidata de tamaño n se chequea contra la tabla de tuplas frecuentes de tamaño $n - 1$ para verificar si la misma puede o no ser frecuente. En algunos casos esta implementación no es eficiente o simplemente no es viable, por ejemplo si los registros pueden tener cientos o miles de ítems la generación de todas las tuplas posibles para cada registro es extremadamente ineficiente. En estos casos tenemos que cambiar la forma en que generamos los candidatos en el algoritmo.

Lo que haremos es generar los candidatos de tamaño n a partir de los candidatos de tamaño $n - 1$ encontrados en la fase anterior. Por ejemplo si tenemos las tuplas frecuentes de tamaño 1:

```
pepsi
carne
coca
papas
pollo
```

Podemos generar todas las tuplas de 2 elementos mediante las combinaciones de los anteriores tomados de a 2:

```
pepsi,carne
pepsi,coca
pepsi,papas
pepsi,pollo
carne,coca
carne,papas
carne,pollo
coca,papas
coca,pollo
papas,pollo
```

En la próxima pasada por cada registro vamos a scanear esta lista y ver si sus elementos aparecen en el registro. Esto puede hacerse simplemente recorriendo el registro y contabilizando cuales son los ítems que aparecen en el mismo

lo cual es mas eficiente que verificar la pertenencia de cada ítem de cada tupla al registro ya que repetiríamos muchas veces la misma operación.

Es decir que si el registro es: {pepsi,carne,tomate} lo recorremos y generamos una tabla de hash que tiene pepsi=1 carne=1 tomate=1 (la cantidad no importa). Luego recorremos cada tupla candidata, por ejemplo {pepsi,carne} y verificamos si ambos están en la tabla, en caso afirmativo sumamos 1 a la cantidad de veces que apareció esta tupla.

Esta versión del algoritmo necesita mas memoria para las tuplas generadas luego de cada paso pero es mas eficiente para recorrer los registros en caso de que estos sean muy grandes.

7.2 PCY

El algoritmo PCY [Park-Chen-Yu] es una simple mejora de la variante de A-Priori que genera las tuplas intermedias, la idea es reducir la cantidad de tuplas candidatas de tamaño 2 que puede ser un número muy grande.

Para lograr esto durante la primer pasada de A-Priori vamos a contabilizar todos los ítems de tamaño 1 pero al mismo tiempo vamos a hashear todas las combinaciones de 2 elementos en una tabla de hash de tamaño fijo. Es decir un counting-filter. Fijaremos la longitud de esta tabla de forma de aprovechar la memoria disponible, cuanto mas grande la tabla menos colisiones y mas útil resulta la misma. Por ejemplo si nuestra tabla tiene 4 millones de posiciones hasheamos cada par de ítems de cada registro, por ejemplo $h(\{\text{carne}, \text{coca}\}) = 931099$. La función de hashing nos devuelve un número entre 0 y 4 millones menos 1. Vamos a la posición indicada por la función de hashing y sumamos 1.

Una vez que tenemos todos los ítems frecuentes de tamaño 1, empezamos a generar las tuplas candidatas de tamaño 2 y al hacerlo las hasheamos y verificamos el contador de las mismas en la tabla de hash, si el contador es menor al soporte entonces la tupla no puede ser frecuente y no la generamos, la descartamos.

Notemos que si el contador es mayor o igual al soporte no necesariamente la tupla es candidata, ya que su contador podría haber sido incrementado también por alguna otra tupla (colisión).

Luego de esta fase el algoritmo continua igual que A-Priori, el truco del counting-filter se usa únicamente entre la primera y la segunda pasada.

Existen muchas variantes de A-Priori basadas en estas ideas, con diferentes tablas de hash y diferentes etapas.

7.3 Algoritmos Aleatorizados

En los casos extremos en los cuales tenemos una enorme cantidad de registros y donde los registros a su vez pueden ser muy grandes es posible que el tiempo o la memoria insumidos por A-Priori o sus variantes sean inviables, la solución

mas eficiente es recurrir a algoritmos aleatorizados que no necesariamente nos dan una solución exacta al problema pero cuya eficiencia podemos controlar. Como siempre a mayor velocidad menor precisión y vice-versa.

7.3.1 Muestreo Aleatorio

El caso mas simple es tomar un muestreo al azar de registros de nuestros datos y correr A-Priori sobre estos registros. Supongamos que tenemos 1 millón de registros y definimos como soporte mínimo a un 1% de los registros es decir que para ser frecuente un ítem tiene que aparecer al menos 10.000 veces.

Imaginemos que tomamos como muestreo un 10% de los registros es decir 100.000 registros. Como hemos dividido los datos en 10 partes nuestro nuevo soporte es 1/10 del soporte anterior. Es decir que en la muestra de 100.000 registros un ítem será frecuente si aparece 1000 veces.

El algoritmo por muestreo puede darnos falsos-positivos y falsos-negativos. Un falso-positivo es un ítem que es frecuente en la muestra pero no lo es en el archivo completo. En nuestro caso un ítem que aparece 1000 veces o mas en la muestra pero menos de 10.000 veces en el archivo completo. Un falso-negativo es un ítem que es frecuente en el archivo completo pero no en la muestra. En nuestro caso un ítem que aparece menos de 1000 veces en la muestra pero 10.000 veces o mas en la totalidad del archivo.

Es muy importante tomar la muestra al azar, el método mas seguro es recorrer cada registro del archivo e incluirlo en la muestra con probabilidad p , en nuestro caso $p = 0.1$ para que la muestra sea un 10% del archivo original.

7.3.2 Algoritmo SON

SON [Savasere,Omiecinski,Navathe] es un algoritmo aleatorizado que hace solo dos pasadas sobre los datos. El resultado de SON no tiene falsos positivos ni falsos negativos. Este algoritmo tiene como ventaja que es fácilmente paralelizable por lo que veremos como implementarlo usando Map-Reduce, pero primero vamos a describir el algoritmo.

Dividimos el archivo original en una cierta cantidad de bloques, por ejemplo nuestro archivo de 1 millón de registros puede dividirse en 10 bloques de 100000 registros cada uno. Luego sobre cada bloque aplicamos A-Priori pero usando el soporte que corresponde, si el soporte original era s y dividimos el archivo en k bloques entonces el soporte en cada bloque será s/k . El resultado es la lista de tuplas frecuentes en cada bloque. Notar que cada bloque produce tuplas de tamaño 1,2,3,etc...

Las tuplas candidatas son la unión de todas las tuplas candidatas de cada bloque. Si una tupla no es candidata en ningún bloque entonces no puede ser frecuente. Veamos esto con nuestro ejemplo. Cada bloque tiene 100.000 registros, y en cada bloque le pedimos a cada tupla que aparezca 1000 veces para ser frecuente. Si una tupla no tiene soporte 1000 en ningúun bloque entonces no puede tener soporte 10000 en el archivo completo. Esto indica que no puede

haber falsos negativos. Pero todavía puede haber falsos positivos es decir tuplas que hayan tenido soporte 1000 o mas en algún bloque pero que no sumen 10000 en el total del archivo. Para eliminar los falsos positivos SON hace una segunda pasada a los datos en donde contamos cuantas veces aparece cada tupla candidata, finalmente filtramos estos resultados y tenemos el resultado final.

SON en Map-Reduce

La primera fase de SON tiene que procesar cada **bloque** del archivo y generar todas las tuplas candidatas. Para eso tenemos que usar la transformación de Spark *mapPartitions* que recibe una función que se aplica a una partición entera es decir a un conjunto de registros. La función recibe un iterable (los registros) y devuelve otro iterable. La función que pasaremos es nada mas ni nada menos que una versión en memoria de A-Priori. El resultado de esta fase es un RDD en donde tenemos como clave la tupla que es frecuente en la partición y como valor un simple 1.

En la fase de Reduce simplemente hacemos un *ReduceByKey* en donde devolvemos la clave ignorando el valor que la acompaña (1). Esto es equivalente a hacer una operación de unión. Las tuplas que fueron frecuentes en mas de una partición quedan una sola vez en el resultado.

El resultado final de la fase de Reduce tiene que poder cargarse en memoria, por ejemplo en una lista.

En la segunda fase hacemos un Map común que procese cada registro del archivo y genere algo de tipo (tupla,1) por cada tupla de la lista de frecuentes que aparezca en el registro.

La fase 2 de reduce hace un *reduceByKey* sumando la cantidad de veces que cada tupla apareció.

Finalmente hacemos un map del resultado de reduce filtrando aquellas tuplas que no cumplen con el soporte pedido.

A continuación veremos la versión Map-Reduce de SON usando Apache Spark:

Vamos a usar como input un archivo de texto en donde cada línea es un registro y los ítems están separados por comas, un fragmento es el siguiente:

```
tomate, pepsi
lechuga, espinaca, acelga
carne, coca, papas
coca, asado, vacio, chorizo, morcilla
pañales, cerveza, salame, queso, aceitunas, papas
carne, coca, papas
pepsi, pollo, papas
...
...
```

El primer paso es leer el archivo y convertirlo en un RDD en donde cada registro es una lista de ítems.

¹ fileName = 'datosapriori.txt'

```

3 itemsRDD = (sc.textFile(fileName, 4))
itemsRDD = itemsRDD.map(lambda x:re.sub(r'[^a-zA-Z0-9]', ' ', x) )
    .map(lambda x:x.split())

```

En este momento itemsRDD es un RDD de Spark en el cual cada registro es una lista de ítems. Notar que hemos usado textFile(filename,4) para cargar el archivo que divide a nuestro archivo en 4 (cuatro) particiones.

Tenemos ahora que llamar a A-Priori para cada partición usando mapPartitions, luego hacemos un reduceByKey para eliminar duplicados y como los registros tienen formato (tupla,1) nos quedamos solo con la clave es decir con los ítems frecuentes.

```

1 itemsRDD1 = itemsRDD.mapPartitions(apriori).reduceByKey(lambda x,y:
x).map(lambda x:x[0])

```

Solamente a efectos de que puedan probar el código si así lo desean listamos a continuación la versión de A-Priori en memoria que hemos usado. Es muy probable que pueda mejorarse bastante el código.

```

1 def filterDictionary(d, support):
2     for item in d.keys():
3         if d[item]<support:
4             del(d[item])
5     return d
6
7 def apriori(listOfBaskets):
8     listOfBaskets=list(listOfBaskets)
9     minsupport = 2
10    frequentItems = dict()
11    result = dict()
12    # First scan all items
13    for basket in listOfBaskets:
14        for item in sorted(basket):
15            if (item,) in frequentItems:
16                frequentItems[(item,)]+=1
17            else:
18                frequentItems[(item,)] = 1
19    # Filter items mantain only individual items with min support
20    frequentItems = filterDictionary(frequentItems, minsupport)
21    order = 2
22    result.update(frequentItems)
23    while(frequentItems):
24        newFrequentItems = dict()
25        for basket in listOfBaskets:
26            all_combinations = itertools.combinations(sorted(basket),
27                order)
28            for combination in all_combinations:
29                subcombination = itertools.combinations(list(
30                    combination),order-1)
31                # All items must be frequent
32                isfrequent = True
33                for item in sorted(subcombination):
34                    if item not in frequentItems:
35                        isfrequent = False
36            if isfrequent:
37                newFrequentItems[combination] = 1
38    result.update(newFrequentItems)
39    return result

```

```

33             if item not in frequentItems:
34                 isfrequent = False
35                 break
36             # if all subcombinations are frequent then add
37             combination
38                 if isfrequent:
39                     if tuple(sorted(combination)) in
40                     newFrequentItems:
41                         newFrequentItems[tuple(sorted(combination))]
42                         ]+=1
43                     else:
44                         newFrequentItems[tuple(sorted(combination))]
45                         ]=1
46                     # Now filter each member of the dictionary by support
47                     newFrequentItems = filterDictionary(newFrequentItems,
48                     minsupport)
49                     frequentItems = newFrequentItems
50                     result.update(frequentItems)
51                     order = order+1
52                     vector = []
53                     for item in result.keys():
54                         vector.append((item,1))
55                     return vector

```

Una vez que tenemos la lista de candidatos nos queda filtrarla en la segunda fase para eliminar falsos positivos. Para hacer esto tomamos el resultado en una lista de tuplas candidatas, que tienen que entrar en memoria y aplicamos un flatMap en donde nos fijamos por cada elemento de la lista si el mismo ocurre en el registro. Luego reduceByKey suma ocurrencias y filter se encarga de dejar únicamente los conjuntos de ítems que son realmente frecuentes.

```

1 listaItemsFrecuentes = itemsRDD1.collect()
2 print itemsRDD.flatMap(filterByList(listaItemsFrecuentes)).
3 reduceByKey(lambda x,y:x+y).filter(lambda x:x[1]>8).collect()

```

La función filterByList es la siguiente:

```

def filterByList(listaFreqs):
    def bar(row):
        ret = []
        # We have to check if the items in listaFreqs are in this
        row
        for item in listaFreqs:
            if set(item).issubset(set(row)):
                ret.append((item,1))
        return ret
    return bar

```

Notemos que esta es una versión particular de Map en la cual le pasamos un parámetro externo (la lista de candidatos), Spark se encargará de replicar esta

lista de candidatos en la memoria de cada equipo que tenga que correr este Map. Para poder pasar un parámetro a Map usamos como truco funciones anidadas. Llamamos a una función que recibe el parámetro externo y esta función llama a una función anidada que recibe el registro como parámetro. Este sencillo truco amerita la preparación de algún trago y un brindis.

El algoritmo SON es importante porque permite correr A-Priori en un ambiente distribuido de forma sencilla. Notemos que en este capítulo usamos un par de trucos nuevos de Spark: mapPartitions y funciones anidadadas para poder pasar un parámetro extra a map.

7.3.3 Algoritmo de Toivonen

El algoritmo de Toivonen es un algoritmo aleatorizado, el resultado que devuelve es exacto es decir que no tiene falsos-positivos ni falsos-negativos. La cantidad de pasadas que el algoritmo hace sobre los datos es variable y puede llegar a funcionar en una sola pasada. Es el único de los algoritmos que vimos que tiene esta propiedad.

El algoritmo va a tomar una muestra de los datos al igual que el algoritmo muestral y va a obtener los ítems frecuentes a partir de esta muestra usando un soporte que es algo menor a lo que sería el soporte proporcional para la muestra, en general un valor entre el 80% y el 90% es adecuado. Por ejemplo si tenemos 1 millón de registros y el soporte es 10.000, si tomamos una muestra de 100.000 registros el soporte sería un número entre 800 y 900 en contraste con 1000 que sería lo que corresponde para el tamaño de la muestra.

Una vez que el algoritmo obtiene los ítems frecuentes a partir de la muestra va a construir lo que llamamos la *frontera negativa* que son todos los conjuntos de ítems que no son frecuentes pero para los cuales todos sus subconjuntos son frecuentes.

A modo de ejemplo supongamos que los ítems son {A,B,C,D,E} y que luego de correr A-priori en memoria sobre la muestra tenemos como frecuentes los siguientes conjuntos de ítems: {A}; {B}; {C}; {D} ; {B,C}; {C,D}. El conjunto vacío se supone frecuente por lo que {E} va a estar en la frontera negativa ya que "E" no es frecuente pero vacío si lo es. Luego tomando de a dos encontramos que {A,B} está en la frontera negativa ya que no es frecuente pero tanto A como B lo son, lo mismo para {A,C}; {A,D} y {B,D}. Ninguna dupla que tenga "E" está en la frontera negativa ya que "E" no es frecuente. A partir de las duplas podemos construir triples pero ninguna tripla puede estar en la frontera negativa ya que para estarlo todas sus duplas deberían ser frecuentes. Notemos que la construcción de la frontera negativa es igual a la construcción de candidatos de A-Priori.

Finalmente hacemos una pasada sobre los datos y contamos la cantidad de veces que aparecen los conjuntos candidatos y los conjuntos en la frontera negativa. Si ningún conjunto en la frontera negativa es frecuente entonces el algoritmo termina y los conjuntos frecuentes son el resultado exacto. Si alguno de los conjuntos en la frontera negativa es frecuente entonces es necesario repetir

el algoritmo a partir de otra muestra aleatoria. El algoritmo podría quedar en un loop infinito pero la probabilidad de que esto pase es prácticamente cero.

Analicemos ahora el funcionamiento del algoritmo. Es evidente que no tenemos falsos positivos ya que realizamos una pasada a todos los datos y contamos cuantas veces aparece cada candidato por lo que nos quedaremos solo con aquellos candidatos que obtuvimos en la muestra y que son realmente frecuentes en el total. Postulamos ahora que si ningún conjunto en la frontera negativa es frecuente entonces todos los conjuntos frecuentes estaban en la muestra.

Para demostrar esto analicemos que pasa si suponemos que ningún conjunto en la frontera negativa es frecuente pero que existe un conjunto frecuente "S" que no fue encontrado en la muestra. Sabemos que si "S" es frecuente en el total entonces todos sus subconjuntos también lo son. Llamemos "T" al subconjunto de "S" de menor cantidad de elementos que no es frecuente en la muestra. Queremos probar que "T" tiene que estar en la frontera negativa. "T" por definición no es frecuente en la muestra pero todos sus subconjuntos tienen que serlo ya que sino existiría otro "T" con menor cantidad de elementos contradiciendo nuestra hipótesis. Por lo tanto "T" está en la frontera negativa lo cual demuestra que el algoritmo solo puede tener falsos positivos si algún conjunto de la frontera negativa es frecuente.

Como el párrafo anterior puede resultar un poco confuso volvamos a nuestro ejemplo en donde tenemos como candidatos a $\{A\}$; $\{B\}$; $\{C\}$; $\{D\}$; $\{B,C\}$ y $\{C,D\}$ y la frontera negativa tiene a: $\{E\}$; $\{A,B\}$; $\{A,C\}$; $\{A,D\}$; $\{B,D\}$. Supongamos que existe un conjunto que es frecuente pero no está en la frontera negativa ni es candidato. Por ejemplo $\{A,B,C\}$, para que $\{A,B,C\}$ sea frecuente todos sus subconjuntos deberían también serlo por lo tanto encontraríamos a $\{A,B\}$ como frecuente en la frontera negativa. En definitiva no existe ningún conjunto de ítems que no sea frecuente en la muestra ni tenga un subconjunto en la frontera negativa. Esto se demuestra fácilmente por inducción.

Para que el algoritmo de Toivonen realice la menor cantidad de pasadas a los datos es necesario darle la mayor cantidad de memoria posible y eso se regula con el umbral que usamos para definir que un conjunto es frecuente en la muestra. Cuanto más chico sea el umbral más memoria hace falta y menos probable es que el algoritmo necesite hacer otra pasada. Notemos que el algoritmo no se refina sino que simplemente se reinicia si necesita otra pasada, es equivalente a hacer un segundo intento.

7.4 Reglas de Asociación

Habiendo estudiado los algoritmos necesarios para encontrar conjuntos de ítems frecuentes dado un cierto soporte analizamos ahora como encontrar a partir de éstos reglas de asociación. Una regla de asociación es de la forma: $A \rightarrow B$ que significa que aquellos registros con el ítem "A" también suelen tener el ítem "B". Puede entenderse la asociación como una implicación. En el caso de un vendedor online la interpretación sería "los que compran A también suelen comprar B".

El soporte es suficiente para encontrar ítems frecuentes pero puede resultar

una métrica insuficiente para reglas de asociación. Por ejemplo si "agua" y "pan" son ítems frecuentes no necesariamente podemos implicar que los que compraron agua suelen comprar pan o los que compraron pan suelen comprar agua, simplemente que ambos ítems aparecen juntos con una cierta frecuencia.

Es por esto que para reglas de asociación agregamos otra métrica que llamamos *confianza* (confidence). La confianza en una regla de tipo $A \rightarrow B$ se define como la cantidad de veces que aparece B en aquellos registros que tienen A .

Por ejemplo supongamos que tenemos 20000 registros si A aparece en 1000 registros y de estos en 100 aparece junto con B el soporte de $A \rightarrow B$ es de $100/20000=0.005$ mientras que la confianza es $100/1000=0.1$. Analicemos ahora los diferentes escenarios posibles que tenemos y un ejemplo basado en datos reales.

7.4.1 Soporte y Confianza en Reglas de Asociación

Soporte Alto, Confianza Alta

Ejemplo: Pan \rightarrow agua. En este caso tenemos que los ítems son frecuentes y que suelen comprarse juntos. Estas reglas no suelen ser muy interesantes porque en general nos indican cosas que ya sabemos, no aportan mucha información.

Soporte Alto, Baja Confianza

Ejemplo: Agua \rightarrow atún. Estos ítems son los suficientemente populares como para ser frecuentes juntos pero no suelen comprarse juntos. No son reglas que nos interesen ya que pueden existir muchísimos casos.

Soporte Bajo, Confianza Baja

Ejemplo: Detergente para lavavajillas \rightarrow fritolín. Los ítems no son frecuentes ni están relacionados. Claramente no estamos interesados en estos casos.

Soporte Bajo, Confianza Alta

Ejemplo: Vodka \rightarrow Jugo de tomate. En este caso los ítems no son muy frecuentes pero están fuertemente relacionados. Estas suelen ser las reglas mas interesantes ya que nos permiten descubrir relaciones fuertes entre ítems poco frecuentes.

7.4.2 Limitaciones del Soporte y Confianza para encontrar asociaciones interesantes

Analicemos ahora la siguiente tabla:

Tenemos un total de 1000 estudiantes y analizamos quienes juegan basket y quienes comen cereal. La regla de asociación juega basket \rightarrow come cereal tiene un 40% de soporte ya que 400 de los 1000 alumnos juegan basket y comen cereal. La confianza es del 66.7% ya que de los 600 que juegan basket 400 comen

	juga basket	no juega	total
come cereal	400	350	750
no come	200	50	250
total	600	400	1000

Table 7.1: Soporte y Confianza en Reglas de Asociación

cereal. Analicemos ahora la regla no juega basket → come cereal. El soporte es del 35% ya que son 350 los alumnos que no juegan basket y comen cereal. La confianza es del 87.5% ya que de los 400 que no juegan al basket 350 comen cereal. Ambas reglas tienen niveles de soporte y confianza altos y sin embargo son contradictorias. Si nos ponemos en el lugar de una firma que quiere hacer publicidad de cereales no sabemos si tenemos que apuntar a los alumnos que juegan basket o a aquellos que no juegan. Es conveniente contar con alguna métrica que nos permita calcular que tan *interesante* es una asociación, vamos a ver una de estas métricas: Lift.

7.4.3 Lift

Vamos a definir la métrica *Lift* como:

$$\text{lift}(B, C) = \frac{c(B \rightarrow C)}{s(C)} = \frac{s(B \cup C)}{s(B)s(C)} \quad (7.1)$$

Donde $s(C)$ y $c(C)$ son soporte y confianza respectivamente. Analicemos el valor de lift en nuestro ejemplo:

$$\text{lift}(\text{basket}, \text{cereal}) = \frac{0.66}{0.75} = 0.8893$$

$$\text{lift}(\text{no basket}, \text{cereal}) = \frac{0.875}{0.075} = 1.16$$

Lift indica que tipo de relación tienen B y C . Cuando el lift es igual a 1 las variables son independientes. Cuando el lift es mayor a 1 existe una correlación positiva entre las variables y cuando el lift es menor a 1 existe una correlación negativa. En nuestro caso jugar al basket está negativamente relacionado con comer cereal mientras que no jugar al basket está positivamente relacionado por lo que nuestra firma de cereales debería apuntar a aquellos alumnos que no juegan al basket.

Lift es una buena métrica para juzgar reglas de asociación pero tiene un problema: es sensible a los valores nulos es decir que los registros en los cuales no aparecen ninguna de las variables involucradas, podemos ver el siguiente ejemplo:

Podemos ver que $B \rightarrow C$ es mucho menos frecuente que $B \rightarrow -C$ y que $-B \rightarrow C$ por lo tanto podemos deducir que la regla $B \rightarrow C$ no debería ser interesante. Sin embargo el valor para $\text{lift}(B, C)$ es 8.44 lo cual indica que las

	B	-B	total
C	100	1000	1100
-C	1000	100000	101000
total	1100	101000	102100

Table 7.2: Soporte y Confianza en Reglas de Asociación

variables están relacionadas positivamente. Esto ocurre porque hay una gran cantidad de registros en donde no ocurren ni B ni C. El problema no solo se da cuando hay muchos registros nulos sino también en casos en los que hay muy pocos nulos.

Para solucionar este problema necesitamos métricas que sean insensibles a los nulos es decir a los registros en donde no ocurre ninguno de los ítems que estamos estudiando.

7.4.4 Métricas invariantes a los valores nulos

Las siguientes métricas son invariantes a los valores nulos, es decir que dan el mismo resultado independientemente de cuántos registros no tienen a ninguno de los ítems involucrados.

Métrica	Definición	Rango
AllConf(A,B)	$\frac{s(A \cup B)}{\max s(A), s(B)}$	[0,1]
Jaccard(A,B)	$\frac{s(A \cup B)}{s(A) + s(B) - s(A \cup B)}$	[0,1]
Cosine(A,B)	$\frac{s(A \cup B)}{\sqrt{s(A)s(B)}}$	[0,1]
Kulczynski(A,B)	$\frac{1}{2} \left(\frac{s(A \cup B)}{s(A)} + \frac{s(A \cup B)}{s(B)} \right)$	[0,1]

Table 7.3: Métricas invariantes a los registros nulos

En general estas métricas funcionan de forma bastante similar y dependiendo de los datos puede ser mejor usar una u otra. Son fórmulas que debemos recordar probar siempre que estemos buscando algún criterio para definir que reglas de asociación son interesantes en un set de datos.

Chapter 8

Reducción de Dimensiones

Either this is madness or it is Hell. -It is neither,calmly replied the voice of the Sphere-, it is Knowledge; it is Three Dimensions

Edwin, A, Abbot, Flatland

8.1 Introducción

En este capítulo desarrollaremos en mas detalle los algoritmos necesarios para manipular la dimensionalidad de los datos que habíamos introducido en el capítulo 1. Los algoritmos para cambiar la dimensionalidad de los datos son fundamentales en Data Science, antes de ver cada uno de ellos repasemos cuáles son los motivos por los cuales puede ser necesario manipular la dimensionalidad de los datos.

8.1.1 Para Combatir la Maldición de la Dimensionalidad

Este puede ser el motivo principal por el cual cambiar la dimensionalidad de los datos, algunos algoritmos funcionan mejor en pocas dimensiones y otros funcionan mejor en muchas dimensiones. Pero para todos los algoritmos si cambiamos levemente la dimensionalidad de los datos ya sea quitando o agregando dimensiones el rendimiento va a cambiar, estudiar si el rendimiento de un algoritmo mejora o empeora al agregar o quitar dimensiones es una tarea muy importante en todo proceso de Data Science.

8.1.2 Por motivos de performance

En muchos casos el algoritmo que queremos usar es inviable para nuestro set de datos por motivos de performance, cuando el factor crítico es la cantidad

de dimensiones en las que se presentan los datos es conveniente reducir las dimensiones de los mismos para poder aplicar nuestro algoritmo.

8.1.3 Como un mecanismo de Feature Engineering

Los algoritmos de reducción de dimensiones sirven muchas veces para reducir el ruido de un set de datos filtrando los features ruidosos y devolviendo un nuevo set de features con menor cantidad de ruido. En otros casos es posible reducir las dimensiones de nuestro set de datos a unas pocas columnas y luego agregar estas columnas al set de datos original, este aumento de dimensiones muchas veces agrega features en donde la señal es muy fuerte en relación al ruido y es frecuente que un algoritmo funcione mejor al agregar estos features con respecto a su funcionamiento con el set de datos original.

8.1.4 Para visualizar los datos

La visualización es uno de los aspectos mas importantes para poder entender un set de datos, el cerebro humano está especialmente preparado para entender datos en dos dimensiones y en menor medida en tres dimensiones. Visualizar datos en mas de tres dimensiones es difícil pero existen algunas técnicas de visualización que lo permiten. Por ejemplo en un plot de puntos que originalmente tiene dos dimensiones podemos usar el color de los puntos o el tamaño de los mismos para agregar dos dimensiones mas, esto es especialmente útil cuando las dimensiones tienen todos significados diferentes.

En general para poder visualizar los datos vamos a usar algoritmos de Manifold Learning que nos permitan representar los datos en dos dimensiones.

La figura 8.1 nos muestra el resultado de aplicar 8 algoritmos de Manifold Learning diferentes a un mismo set de datos. El set de datos es una superficie con forma de "S" en un espacio tridimensional, podemos observar que se trata de un manifold de 2 dimensiones embebido en un espacio de tres dimensiones. Los puntos fueron coloreados de acuerdo a la zona de la "S" en la que están, un buen algoritmo de Manifold Learning debería mantener a los punto que están cerca en los datos originales como vecinos en su representación bidimensional. Como puede verse diferentes algoritmos logran esto con diferentes representaciones.

Los algoritmos para manipular la dimensionalidad de los datos pueden dividirse en dos grandes grupos: Métodos lineales y métodos no-lineales. Los métodos lineales por excelencia son la SVD y PCA que en realidad son un mismo algoritmo. Los métodos no-lineales son mas variados y se denominan también algoritmos para "Manifold Learning".

8.2 SVD

La SVD (Singular Value Decomposition) es la herramienta mas importante para la reducción de dimensiones de un set de datos. Vamos a suponer que nuestro set de datos es una matriz de m filas y n columnas o dimensiones.

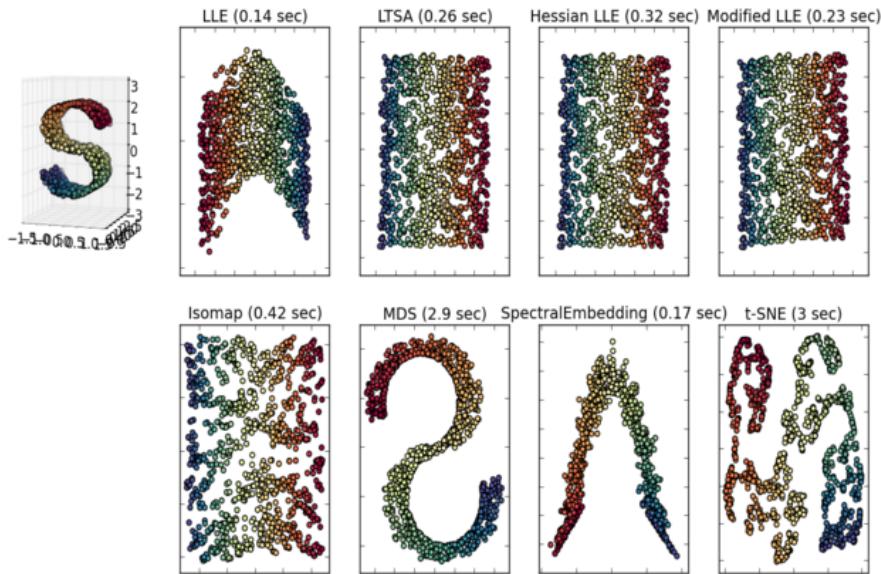


Figure 8.1: Algoritmos de Manifold Learning

Cualquier dato en n dimensiones es una hiper-elipse con n ejes, si a este dato le aplicamos una transformación lineal lo que obtenemos es otra hiper-elipse.

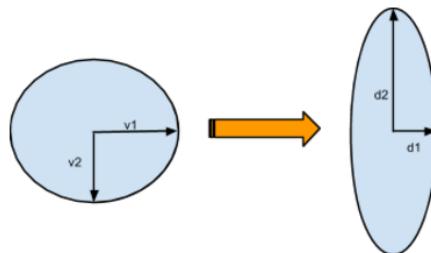


Figure 8.2: SVD

Cualquier dato en dos dimensiones puede expresarse como una transformación lineal aplicada a un círculo de radio 1. Los ejes de la elipse surgen de multiplicar a los ejes del círculo por un par de constantes y luego eventualmente podemos rotar la elipse para obtener cualquiera de todas las elipses posibles en dos dimensiones.

Un círculo está definido por una base de vectores unitarios $v_1..v_n$ por lo tanto podemos escribir la siguiente ecuación:

$$AV = U\Sigma$$

Lo que esta ecuación nos dice es que si aplicamos una matriz cualquiera A a una base unitaria V lo que obtenemos es una hiper-elipse que podemos expresar como el producto de una base U por una matriz diagonal Σ que nos indica la forma de estirar o contraer los ejes del círculo para obtener los ejes de la elipse.

Multiplicando a ambos lados por V^{-1}

$$AVV^{-1} = U\Sigma V^{-1}$$

$VV^{-1} = I$ y además como V es una matriz unitaria su inversa es igual a su transpuesta.

$$A = U\Sigma V^t$$

Esta es la definición de la SVD, toda matriz se puede descomponer entonces en el producto de tres matrices: U , Σ y V . El cálculo de la SVD de una matriz es una tarea relativamente sencilla que se limita al cálculo de autovectores y autovalores.

Sea A una matriz cualquiera:

$$\begin{aligned} A^t A &= (U\Sigma V^t)^t (U\Sigma V^t) \\ A^t A &= (V\Sigma U^t U\Sigma V^t) \\ A^t A &= V\Sigma^2 V^t \\ A^t A V &= V\Sigma^2 V^t V \\ A^t A V &= V\Sigma^2 \end{aligned}$$

Donde V son los autovectores y Σ^2 son autovalores de $A^t A V$ Y de la misma forma:

$$\begin{aligned} A A^t &= (U\Sigma V^t)(U\Sigma V^t)^t \\ A A^t &= (U\Sigma V^t V\Sigma U^t) \\ A A^t &= U\Sigma^2 U^t \\ A A^t U &= U\Sigma^2 U^t U \\ A A^t &= U\Sigma^2 \end{aligned}$$

Donde U son los autovectores y Σ^2 son autovalores de $A A^t$
Por ejemplo sea la siguiente matriz:

$$\begin{pmatrix} 3 & 0 \\ 0 & -2 \end{pmatrix}$$

Calculamos $A^t A$

$$A^T A = \begin{pmatrix} 9 & 0 \\ 0 & 4 \end{pmatrix}, \lambda_1 = 9, \lambda_2 = 4 \Rightarrow \sigma_1 = 3, \sigma_2 = 2$$

$$A^A = \begin{pmatrix} 9 & 0 \\ 0 & 4 \end{pmatrix}, A^t A U = U \begin{pmatrix} 9 & 0 \\ 0 & 4 \end{pmatrix} \Rightarrow U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$A = U \Sigma V^t = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

8.2.1 Interpretación algebraica de la SVD (muy importante!)

Veamos una interpretación algebraica de la SVD que es fundamental para entender varios conceptos importantes en el mundo de Data Science. $A_{m \times n}$ es nuestra matriz de datos. $U_{m \times r}$ es una base ortogonal en donde r es el rango de la matriz A . $\Sigma_{r \times r}$ es una matriz diagonal con los valores singulares ordenados de mayor a menor: $\sigma_1, \sigma_2 \dots \sigma_n$ y finalmente $V_{n \times r}$ es otra matriz ortogonal.

Observemos que V^t al ser una matriz de $r \times n$ tiene la misma dimensionalidad que nuestra matriz original, por lo tanto V es una **base** de r vectores en n dimensiones cada uno. $\Sigma_{r \times r}$ nos da un escalar por el cual multiplicar cada uno de los vectores de la base, como Σ tiene los valores singulares ordenados de mayor nos determina la importancia de los vectores de la base V , el primer vector de V (la primera columna) es el mas importante de la base, el segundo es el segundo mas importante y así sucesivamente. Finalmente la matriz $U_{m \times r}$ tiene tantos vectores como datos tenemos y cada vector tiene dimensionalidad igual a la cantidad de vectores en la base V por lo tanto U indica los coeficientes por los cuales tenemos que multiplicar a cada vector de la base V para reconstruir cada uno de los puntos de nuestro set de datos.

Dado un punto cualquiera de nuestro set de datos por ejemplo A_i lo podemos expresar como una combinación lineal de vectores de la base V de la forma:

$$A_i = U_i * \Sigma * V$$

Mas adelante vamos a ver muchos métodos mas por los cuales podemos descomponer una matriz de datos en el producto de dos matrices, en todos los casos la primera matriz será una matriz de coeficientes y la segunda será una base, diferentes bases nos darán diferentes formas de aproximar la reconstrucción de nuestros datos.

8.2.2 Aproximación de Rango k

Hemos visto que la matriz V es una base en donde los vectores de la base están ordenados por su grado de importancia, la primera columna de V es el vector mas importante de la base, etc. Podemos pensar entonces en aproximar

a nuestra matriz A usando solamente los primeros k vectores de la base V , a esto lo llamamos una aproximación de rango k a nuestra matriz original.

Para aproximar la matriz original con rango k lo que hacemos es tomar las primeras k columnas de U , las primeras k filas y columnas de Σ y las primeras k columnas de V (las primeras k filas de V^t).

Veamos un ejemplo de esto:

Sea:

$$A = \begin{pmatrix} 3 & 2 & 4 & 4 \\ 1 & 0 & 0 & 3 \\ 3 & 1 & 2 & 2 \end{pmatrix}$$

$$[U, \Sigma, V] = SVD(A)$$

$$U = \begin{pmatrix} 0.8138 & 0.2143 & -0.5402 \\ 0.2931 & -0.9540 & 0.0632 \\ 0.5018 & 0.2097 & 0.8392 \end{pmatrix}$$

$$\Sigma = \begin{pmatrix} 8.1884 & 0 & 0 & 0 \\ 0 & 2.1568 & 0 & 0 \\ 0 & 0 & 1.1395 & 0 \end{pmatrix}$$

$$V = \begin{pmatrix} 0.5178 & 0.1475 & 0.8427 & 0 \\ 0.2601 & 0.2960 & -0.2116 & 0.8944 \\ 0.5201 & 0.5919 & -0.4232 & -0.4472 \\ 0.6275 & -0.7350 & -0.2569 & 0 \end{pmatrix}$$

Podemos aproximar nuestra matriz original usando 1, 2 o 3 vectores de V

$$A_3 = \begin{pmatrix} 0.8138 & 0.2143 & -0.5402 \\ 0.2931 & -0.9540 & 0.0632 \\ 0.5018 & 0.2097 & 0.8392 \end{pmatrix} \begin{pmatrix} 8.1884 & 0 & 0 \\ 0 & 2.1568 & 0 \\ 0 & 0 & 1.1395 \end{pmatrix} \begin{pmatrix} 0.5178 & 0.1475 & 0.8427 \\ 0.2601 & 0.2960 & -0.2116 \\ 0.5201 & 0.5919 & -0.4232 \\ 0.6275 & -0.7350 & -0.2569 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} 3 & 2 & 4 & 4 \\ 1 & 0 & 0 & 3 \\ 3 & 1 & 2 & 2 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} 0.8138 & 0.2143 \\ 0.2931 & -0.9540 \\ 0.5018 & 0.2097 \end{pmatrix} \begin{pmatrix} 8.1884 & 0 \\ 0 & 2.1568 \end{pmatrix} \begin{pmatrix} 0.5178 & 0.1475 \\ 0.2601 & 0.2960 \\ 0.5201 & 0.5919 \\ 0.6275 & -0.7350 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} 3.51 & 1.87 & 3.73 & 3.84 \\ 0.93 & 0.01 & 0.03 & 3.01 \\ 2.19 & 1.20 & 2.40 & 2.24 \end{pmatrix}$$

$$A_1 = \begin{pmatrix} 0.8138 \\ 0.2931 \\ 0.5018 \end{pmatrix} (8.1884) \begin{pmatrix} 0.5178 \\ 0.2601 \\ 0.5201 \\ 0.6275 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} 3.45 & 1.73 & 3.46 & 4.18 \\ 1.24 & 0.62 & 1.25 & 1.5 \\ 2.12 & 1.06 & 2.13 & 2.57 \end{pmatrix}$$

La aproximación de rango 3 en realidad es igual a nuestra matriz original, esto es lógico ya que nuestra matriz es de rango 2. La aproximación de rango 2, usando solo dos vectores de la base V es muy parecida a la matriz original, la aproximación de rango 1 no es tan exacta pero tenemos que tener en cuenta que estamos escribiendo los 4 vectores de nuestra matriz original como una combinación lineal de un único vector: $v = [0.5178, 0.2601, 0.5201, 0.6275]$

Se puede demostrar que la SVD nos da la **mejor** aproximación de rango k posible a la matriz original. En donde "mejor" significa que tiene la menor diferencia con la matriz original de acuerdo a la *norma de Frobenius* que es simplemente la sumatoria de cada elemento de la matriz al cuadrado. Es decir que la SVD minimiza la suma de cuadrados de la diferencia entre la matriz original y su aproximación. Esto se demuestra en el siguiente teorema:

Theorem 11 (Eckert-Young) *Sea $A = U\Sigma V^t$ donde U y V son matrices unitarias y Σ es una matriz diagonal con valores ($\sigma_n \leq \sigma_{n-1} \leq \dots \leq \sigma_1$). La i -ésima columna de U es u_i y la i -ésima fila de V^t es v_i^t*

Queremos demostrar que la aproximación de rango k que minimiza $\|A - A^k\|_F$ es:

$$A^k = \sum_{i=1}^k u_i \sigma_i v_i^t$$

Para cualquier matriz B de igual tamaño que A definimos la matriz D de la forma:

$$D = U^t B V \Rightarrow B = U D V^t$$

Entonces:

$$\begin{aligned} \|A - B\|_F^2 &= \|U\Sigma V^t - UDV^t\|_F^2 \\ \|A - B\|_F^2 &= \|\Sigma - D\|_F^2 \\ \|A - B\|_F^2 &= \sum_i |\sigma_i - D_{ii}|^2 + \sum_{i \neq j} |D_{ij}|^2 \\ &\geq \sum_i |\sigma_i - D_{ii}|^2 \geq \sum_{i > k} |\sigma_i - D_{ii}|^2 \end{aligned}$$

Por lo tanto la matriz D de rango k que minimiza $\|A - B\|_F^2$ es la matriz diagonal con elementos $(\sigma_1, \sigma_2, \dots, \sigma_n)$. Y entonces B es igual a A^k

8.2.3 Usando la SVD para aproximar la dimensionalidad intrínseca de los datos

La matriz Σ que contiene los valores singulares ordenados de mayor a menor nos da varias pistas sobre la dimensionalidad intrínseca de nuestros datos, si definimos la *energía* de la matriz como la sumatoria de los valores singulares al cuadrado entonces podemos calcular que porcentaje de energía mantenemos usando los k valores singulares más significativos que equivalen a los primeros k vectores de la base V .

Consideremos la siguiente matriz de 7x7:

$$A = \begin{pmatrix} 1 & 9 & 0 & 7 & -2 & 7 & 6 \\ 9 & 6 & 3 & 4 & 1 & 0 & 0 \\ -1 & -1 & -2 & 9 & 0 & 8 & 7 \\ 8 & 2 & 0 & 0 & 0 & -2 & -2 \\ 7 & 6 & 0 & 1 & 0 & 1 & 0 \\ -1 & 0 & 3 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 2 & -1 & 2 & 2 \end{pmatrix}$$

Si hacemos un plot de los valores singulares de la SVD de la matriz podemos en general tener una buena idea sobre la dimensionalidad intrínseca de nuestros datos.

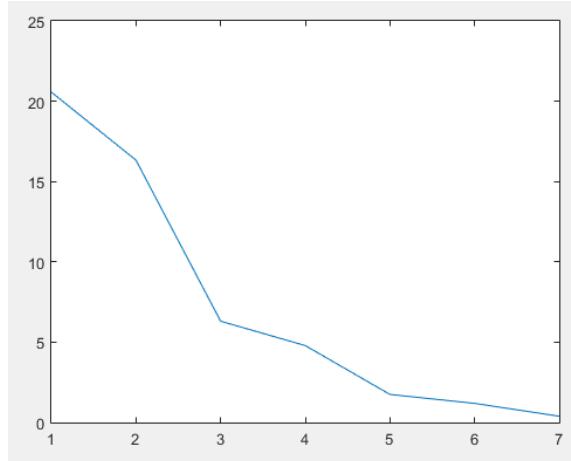


Figure 8.3: Valores Singulares de A

En nuestro ejemplo podemos ver que hay dos "codos" en el plot de los valores singulares, uno en 3 y otro en 5, por lo tanto podemos decir que la dimensionalidad intrínseca de nuestros datos puede ser 3,4 o 5. De lo que podemos estar prácticamente seguros es que descartando los dos últimos valores singulares perdemos muy poco, haciendo las cuentas vemos que con 5 dimensiones conservamos un 99.79% de la energía en la matriz original!

Cantidad de Valores Singulares	Energía
1	55.88
2	91.11
3	96.36
4	99.39
5	99.79
6	99.98

Table 8.1: Eligiendo k

Sorprendentemente con solo 2 vectores en la base V podemos capturar un 91% de la energía de la matriz y con 4 o mas vectores prácticamente capturamos la totalidad. Podemos afirmar entonces que tiene sentido representar a este set de datos en particular en 4 dimensiones y que una representación en dos dimensiones, es decir en el plano es aceptable.

8.2.4 Reducción de dimensiones usando la SVD

Una vez que hemos determinado a que dimensionalidad queremos reducir nuestro set de datos el proceso usando la SVD es muy simple: solo tenemos que quedarnos con las primeras k columnas de la matriz U , esta matriz tiene tantas filas como datos tenemos y tantas columnas como vectores hay en la base, por lo tanto si queremos representar a nuestros datos en una base reducida a k vectores las primeras k columnas de U tienen los coeficientes que necesitamos y sirven como representación de los datos en nuestro nuevo espacio.

8.2.5 Compresión de Imágenes usando la SVD

A modo de ejemplo del poder representativo de la SVD vamos a mostrar como podríamos usar la SVD para comprimir una imagen. Tenemos una imagen de la galaxia de Andrómeda de 127x350 píxeles es decir un total de 44450 píxeles. Podemos tratar ahora de obtener la aproximación de rango k de la matriz y luego simplemente recalcular la imagen como el producto de $U * \Sigma * V$. Los resultados son los siguientes:

En el caso en el cuál usamos, por ejemplo $k = 50$ tenemos que nuestras matrices son de 127x50 y 50x350 esto nos da un total de 23850 píxeles es decir un 53% de la imagen original, y podemos ver que con este nivel de compresión tan extremo la galaxia puede distinguirse bien.

En el caso extremo podemos llegar hasta $k = 2$ en el cual representamos a la imagen en un base de solamente dos dimensiones lo cual nos da una versión muy distorsionada de la galaxia en donde solo retenemos su forma ovalada y el núcleo brillante de la misma. Pero cuando $k = 2$ estamos reteniendo solo un 2% de los píxeles de la imagen original.

Podríamos pensar también en aplicar la SVD para retener la resolución de una imagen pero comprimir sus colores. En una imagen color cada píxel está

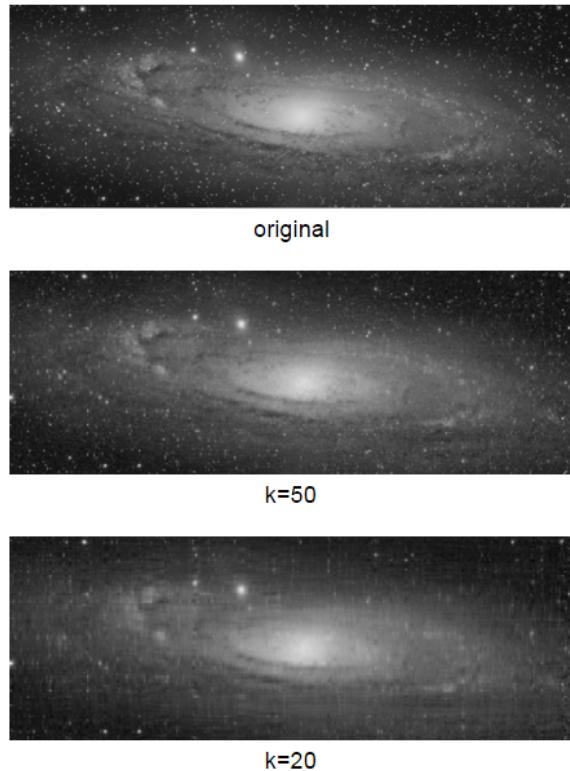


Figure 8.4: Andrómeda via SVD

representado por tres canales: R,G,B lo cual nos da una profundidad de color total de 24 bits. Para esto podemos tomar a nuestro set de datos como una matriz de $3 \times p$ en donde p es la cantidad de pixeles de la imagen.

Aplicamos la SVD y nos quedamos con una representación de rango k de la imagen ahora cada punto de nuestra imagen se ve reducido a alguno de k puntos, es decir las k combinaciones de colores mas representativas de la imagen.

Podemos entonces representar a la imagen en una gama de colores menores ya que en lugar de una cantidad de p colores potencialmente diferentes ahora solo tenemos k entonces podemos almacenar una tabla con los valores *RGB* de estos colores y asignarle a cada pixel un número de entrada en esta tabla, esto es similar a lo que hace el algoritmo JPG y está basado en el hecho de que un ser humano no suele distinguir toda la gama de colores de 24 bits.

Si reducimos los colores a, por ejemplo 1024 colores representativos entonces cada pixel necesita $\log_2 1024 = 10$ bits lo cual implica una compresión de mas del 50% con respecto a los 24 bits originales.

La SVD no es un mecanismo adecuado para comprimir imágenes ya que existen algoritmos específicos para imágenes mucho mejores como JPG. Sin em-

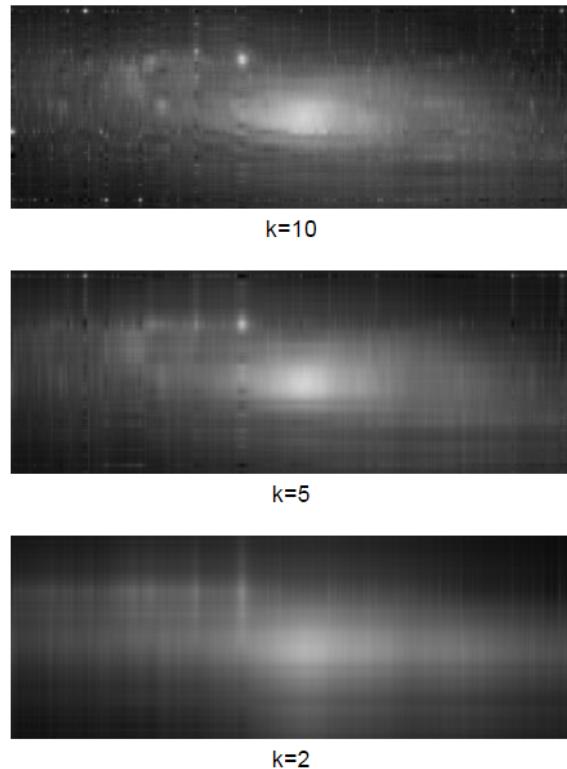


Figure 8.5: Andrómeda via SVD

bargo aplicar SVD a imágenes nos permite una descomposición muy interesante de las mismas e ilustra el funcionamiento del algoritmo para aproximar datos en dimensiones reducidas.

8.2.6 Eigenfaces

Vamos a aplicar la SVD a un set de datos que tiene 400 imágenes de 64x64 píxeles cada una, es decir que tenemos una matriz de 400x4096 en donde cada fila es una imagen y cada columna (dimensión es un píxel).

Aprovechando que tenemos las imágenes como columnas vamos a calcular la SVD y quedarnos con 16 columnas como las 16 imágenes más representativas. Hay que tener en cuenta que estas 16 imágenes que surgen de la reducción de imágenes vía SVD no son 16 imágenes seleccionadas entre las originales sino que son 16 combinaciones lineales de todas las imágenes, las 16 combinaciones lineales más representativas.

Podemos ver que algunos rostros están en negativo, esto es admisible porque en la SVD podemos usar coordenadas negativas para alguno de los vectores. Como

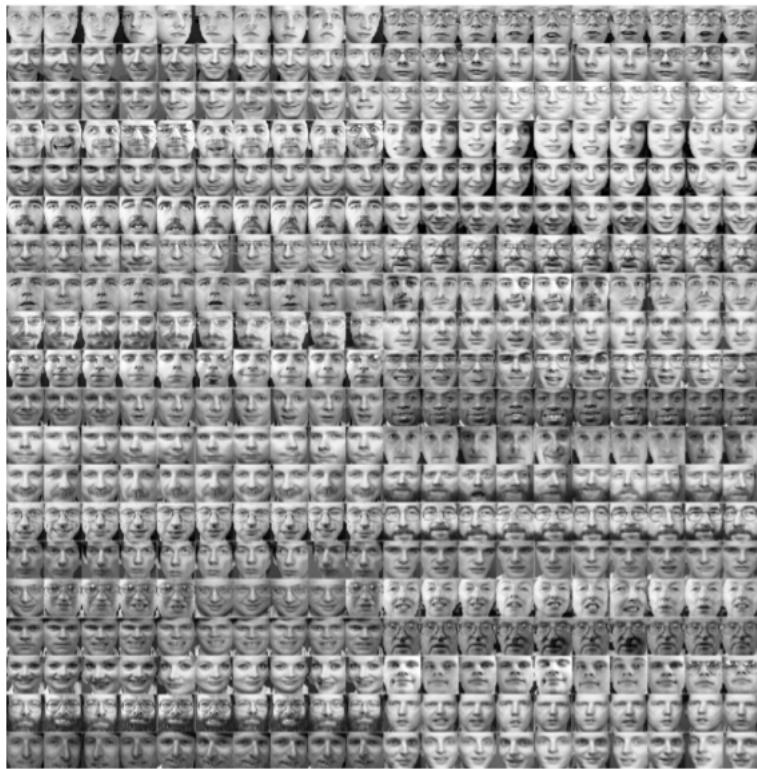


Figure 8.6: Set de Datos de Rostros aparentemente humanos

el primer rostro está en negativo esperamos que la primer coordenada de la matriz "U" esté dominada por valores negativos.

Este algoritmo que hemos usado para comprimir la cantidad de imágenes en un set basado en la SVD se denomina “Eigenpictures” y cuando se aplica a rostros se denomina “Eigenfaces” el algoritmo puede usarse para reconocimiento rápido de rostros de la siguiente manera: Primero calculamos los “ k ” eigenfaces por el método que presentamos recién. Luego por cada rostro a almacenar y que queremos reconocer almacenamos la distancia euclíadiana del mismo a cada uno de los “eigenfaces” en nuestro caso serían en total 16 distancias.

De esta forma cada rostro queda almacenado en una base de datos como un simple vector de 16 componentes. Cuando queremos reconocer un rostro nuevo calculamos sus 16 componentes y buscamos en nuestra base de datos a ver si existe un rostro cuya semejanza en los 16 factores sea muy similar al que queremos reconocer, en base a esto decidimos si tenemos un rostro que ya hemos capturado, un rostro nuevo o una imagen que no es un rostro. Esto último lo podemos calcular mediante la diferencia entre los “ k ” valores de la nueva imagen y un promedio de los eigenfaces.



Figure 8.7: Las 16 EigenFaces mas representativas

8.2.7 SVD en la práctica

En muchos lenguajes y bibliotecas existen rutinas numéricas muy eficientes para calcular la SVD de una matriz para un cierto rango k . Estos procedimientos se basan en el algoritmo de Lanczos que permite de forma iterativa ir obteniendo los autovalores mas significativos de una matriz, como solo queremos k el procedimiento puede ser muy eficiente incluso para matrices muy grandes. Esto es bueno ya que permite aplicar la SVD a datos masivos lo cual es muy importante ya que la SVD es el principal algoritmo para reducir las dimensiones de un conjunto de datos.

8.2.8 Agregando nuevos datos en el nuevo espacio

Una vez que hemos calculado la SVD y reducido nuestro set de datos, tenemos el potencial problema de necesitar agregar nuevos datos, estos nuevos datos van a venir en la dimensionalidad original de los datos. Agregar los datos nuevos a los originales y re-calcular la SVD funciona pero es un proceso potencialmente costoso. Para evitarlo podemos simplemente "plegar" los datos nuevos a nuestro nuevo espacio de dimensiones.

Partimos de:

$$A = U\Sigma V^t$$

Y queremos despejar U que es la matriz reducida en dimensiones, en el caso de un solo dato será de 1xn pero podríamos plegar tantos datos como quisieramos.

$$\begin{aligned} AV &= U\Sigma V^T V = U\Sigma \\ AV\Sigma^{-1} &= U\Sigma\Sigma^{-1} = U \end{aligned}$$

Entonces una vez que tenemos nuevos datos (A) para plegarlos al nuevo espacio de dimensiones tenemos que multiplicarlos por V y luego por Σ^{-1} . De esta forma obtenemos su representación dentro de nuestro nuevo espacio.

Debemos notar que al plegar nuevos vectores al espacio reducido en dimensiones estamos agregando datos que no fueron usados para el cálculo de la SVD, por lo tanto cuando la cantidad de datos agregados supere un cierto umbral la SVD que calculamos originalmente pierde precisión con respecto a su capacidad de aproximar los datos originales en dimensiones reducidas. Por lo tanto si se necesita agregar muchos datos puede ser conveniente cada cierto tiempo reconstruir la SVD por completo.

8.3 PCA: Principal Component Analysis

PCA es uno de los algoritmos mas usados en el mundo de Data Science y vamos a demostrar que es idéntico a la SVD por lo que, irónicamente, nadie debería usarlo, pero dejemos esto para mas adelante y expliquemos la idea y el funcionamiento de PCA.

EL gráfico nos muestra una nube de puntos en dos dimensiones, la idea de PCA es encontrar las "direcciones" principales de los datos es decir aquellas direcciones sobre las cuales podemos proyectar los datos reteniendo su variabilidad. En la figura 8.8 hay dos direcciones principales y el eje mas largo es aquel que nos daría la mayor variabilidad si proyectáramos los datos sobre el mismo.

En concreto PCA ajusta una hiper-elipse a los datos y nos devuelve los ejes de la misma. Estos son los que llamamos *componentes principales*, luego podemos usar estos componentes para proyectar nuestros datos y reducir la dimensionalidad de los mismos.

Una primera observación es que los datos deben estar centrados en el origen para que los componentes principales tengan sentido, este es el primer paso al aplicar PCA, centrar todos los datos de forma tal que el promedio de cada columna sea cero. Esto se logra normalizando los datos restando el promedio de cada columna. Opcionalmente podemos dividir los datos por la desviación standard de cada columna para que todas las columnas tengan promedio cero y desviación standard igual a 1. Luego de centrar los datos es necesario calcular la matriz de covarianza de los mismos que se calcula de la forma:

$$cov(X) = \frac{1}{n-1} X^t X$$

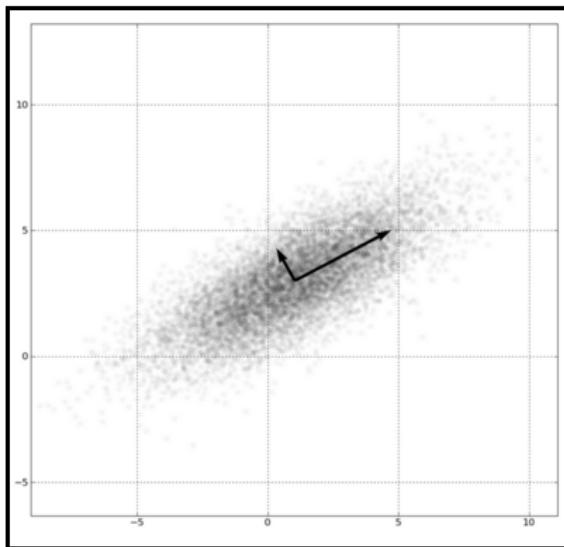


Figure 8.8: PCA

Notar que si los datos no están normalizados la definición de matriz de covarianza es diferente.

La matriz de covarianza es simétrica y en cada celda nos indica la covarianza entre las columnas i y j de nuestra matriz original. Cuando la covarianza es cercana a 0 las columnas son independientes y cuando la covarianza es cercana a 1 o -1 las columnas son dependientes. En el caso concreto de covarianza igual a 1 o -1 una columna es una combinación lineal de la otra y esto quiere decir que una de ellas es redundante.

En el set de datos Iris que tiene 4 dimensiones la matriz de covarianza es la siguiente:

$$\text{coviris} = \begin{pmatrix} 1 & 0.9559 & -0.3670 & 0.8136 \\ 0.9559 & 1 & -0.4354 & 0.8709 \\ -0.3670 & -0.4354 & 1 & -0.1153 \\ 0.8136 & 0.8709 & -0.1153 & 1 \end{pmatrix}$$

En la matriz de covarianza podemos aprender que hay una relación muy fuerte entre las columnas 1 y 2 de nuestros datos es decir el largo y alto de los pétalos de las flores. Esto tiene sentido ya que en una flor es lógico que si los pétalos son grandes entonces el largo del mismo sea proporcional a su altura.

Podemos hacer una matriz de plots de cada columna de nuestro set de datos contra cada otra:

En el plot vemos la relación prácticamente lineal que existe entre las columnas 1 y 2. También hay una relación fuerte entre las columnas 1 y 4 y entre

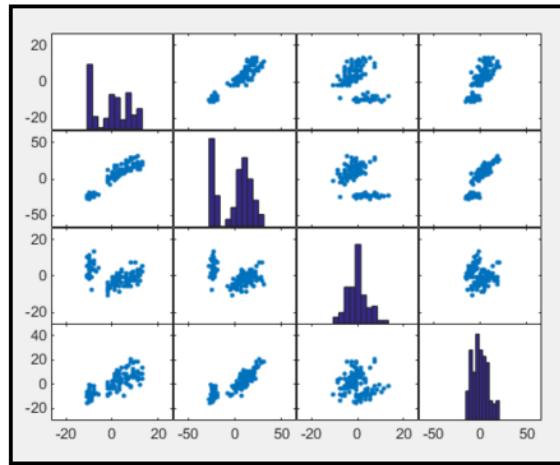


Figure 8.9: PCA para Iris

las columnas 2 y 4. Esto concuerda con los valores que vimos en la matriz de covarianza (0.81 y 0.87 respectivamente).

Para obtener los componentes principales simplemente calculamos los autovalores y autovectores de la matriz de covarianza. En el caso de Iris tenemos:

$$V = \begin{pmatrix} 0.4965 & -0.6565 & 0.0680 & 0.5638 \\ -0.8065 & -0.1090 & 0.0209 & 0.5807 \\ -0.1390 & -0.2431 & 0.9207 & -0.2718 \\ 0.2894 & 0.7057 & 0.3837 & 0.5206 \end{pmatrix} S = \begin{pmatrix} 0.0241 & 0 & 0 & 0 \\ 0 & 0.1483 & 0 & 0 \\ 0 & 0 & 0.9150 & 0 \\ 0 & 0 & 0 & 2.9127 \end{pmatrix}$$

La matriz V tiene los componentes principales y el grado de importancia de los mismos está dado por los elementos de los autovalores (S) aunque en este caso no están ordenados de mayor a menor como en la SVD.

Para finalizar esta sección vamos a demostrar que la base encontrada por PCA es en realidad la misma base que la de la SVD de la matriz asumiendo que la misma ha sido normalizada.

8.3.1 PCA=SVD

Theorem 12 (PCA=SVD) *La matriz de covarianza es $X^t X$, como esta matriz es simétrica es diagonalizable y los autovectores pueden normalizarse de forma tal que sean ortonormales:*

$$X^t X = W D W^t$$

Donde W tiene los autovectores y D los autovalores de la matriz de covarianza.

Por otro lado aplicando la SVD a la matriz X tenemos:

$$X = U\Sigma V^t$$

Si construimos la matriz de covarianza para la descomposición de la SVD tenemos:

$$\begin{aligned} X^t X &= (U\Sigma V^t)^t (U\Sigma V^t) \\ X^t X &= (V\Sigma U^t U\Sigma V^t) \\ X^t X &= V\Sigma^2 V^t \end{aligned}$$

Aquí podemos ver que hay una equivalencia directa por lo tanto $V=W$ y $D=\Sigma^2$. Por lo tanto los autovectores de la matriz de covarianza son iguales a la base de la SVD de la matriz normalizada.

8.3.2 Conclusión

Para concluir debemos señalar que calcular la SVD es en general mas eficiente y numéricamente mas estable que PCA dado que no hace falta construir la matriz de covarianza, es por este motivo que como método genérico lineal para reducción de dimensiones preferimos siempre usar la SVD de la matriz de datos.

8.4 MDS: Multidimensional Scaling

Vamos a explicar el algoritmo llamado *multidimensional scaling* (MDS) a partir de un problema muy simple: Tenemos una matriz con las distancias entre diez ciudades que desconocemos, el objetivo del algoritmo es encontrar coordenadas en el plano para estas diez ciudades de forma tal que las distancias entre las mismas se aproximen lo mas posible a las distancias de nuestra matriz.

Es decir que dada una matriz de distancias entre puntos queremos hallar los puntos. La dimensionalidad de los puntos en nuestro problema la hemos fijado en 2, mas adelante veremos como resolver también la dimensionalidad.

Llámemos D a nuestra matriz de distancias entre n puntos, D es de $n \times n$. Llámemos X de $n \times q$ la matriz con los puntos que desconocemos (en nuestro problema $q = 2$)

Definimos una matriz B :

$$B = XX^t$$

Y sea M una matriz ortogonal ($MM^t = I$) entonces:

$$B = (XM)(XM)^t$$

Los elementos de B están dados por:

$$b_{ij} = \sum_{k=1}^q X_{ik} X_{jk}$$

0	587	1212	701	1936	604	748	2139	2182	543
587	0	920	940	1745	1188	713	1858	1737	597
1212	920	0	879	831	1726	1631	949	1021	1494
701	940	879	0	1374	968	1420	1645	1891	1220
1936	1745	831	1374	0	2339	2451	347	959	2300
604	1188	1726	968	2339	0	1092	2594	2734	923
748	713	1631	1420	2451	1092	0	2571	2408	205
2139	1858	949	1645	347	2594	2571	0	678	2442
2182	1737	1021	1891	959	2734	2408	678	0	2329
543	597	1494	1220	2300	923	205	2442	2329	0

Figure 8.10: Tabla de Distancias Entre Ciudades Desconocidas

Entonces el cuadrado de las distancias entre las filas de X puede escribirse como:

$$D_{ij}^2 = b_{ii} + b_{jj} - 2b_{ij}$$

Si podemos obtener los b_{ij} a partir de los D_{ij} entonces podemos encontrar a X a partir de B .

Vamos a introducir como restricción que las filas y columnas de X estén centradas es decir que tengan promedio cero.

$$\sum_{i=1}^n X_{ik} = 0$$

Esto implica que la suma de términos en cualquier fila de B debe ser cero:

$$\sum_{j=1}^n b_{ij} = \sum_{j=1}^n \sum_{k=1}^q X_{ik} X_{jk} = \sum_{k=1}^q X_{ik} (\sum_{j=1}^n X_{jk})$$

Sea T la traza de B . Para obtener B a partir de D notemos que:

$$\sum_{i=1}^n d_{ij}^2 = T + nb_{jj}$$

$$\sum_{j=1}^n d_{ij}^2 = T + nb_{ii}$$

$$\sum_{i=1}^n \sum_{j=1}^n d_{ij}^2 = 2nT$$

Podemos ahora calcular los elementos de B en función de D :

$$b_{ij} = -\frac{1}{2}(d_{ij}^2 - d_i^2 - d_j^2 + d^2)$$

donde:

$$d_i^2 = \frac{\sum_{j=1}^n d_{ij}^2}{n}$$

$$d_j^2 = \frac{\sum_{i=1}^n d_{ij}^2}{n}$$

$$d^2 = \frac{\sum_{i=1}^n \sum_{j=1}^n d_{ij}^2}{n}$$

Podemos entonces escribir a B de la forma:

$$B = V\Lambda V^t$$

Donde Λ es la matriz diagonal con los autovalores de B y V es la matriz con los autovectores de B como columnas.

Ahora como D surge de una matriz de nxq entonces el rango de D es q (suponiendo $q < n$). Es decir que los últimos $n-q$ autovalores deberían ser cero. Entonces escribimos a B como:

$$B = V^* \Lambda^* V^{*t}$$

Donde V^* es la matriz con los primeros q autovectores de D y Λ^* es la matriz con los primeros q autovalores.

Y tenemos entonces una solución para X :

$$X = V^* \Lambda^{1/2}$$

Y esto nos da un algoritmo para calcular MDS:

1. Elevar la matriz de distancias al cuadrado
2. Centrar la matriz para que las filas y columnas tengan promedio cero
3. Calcular la SVD de la matriz
4. Las primeras q columnas de U nos da las coordenadas para nuestros puntos.

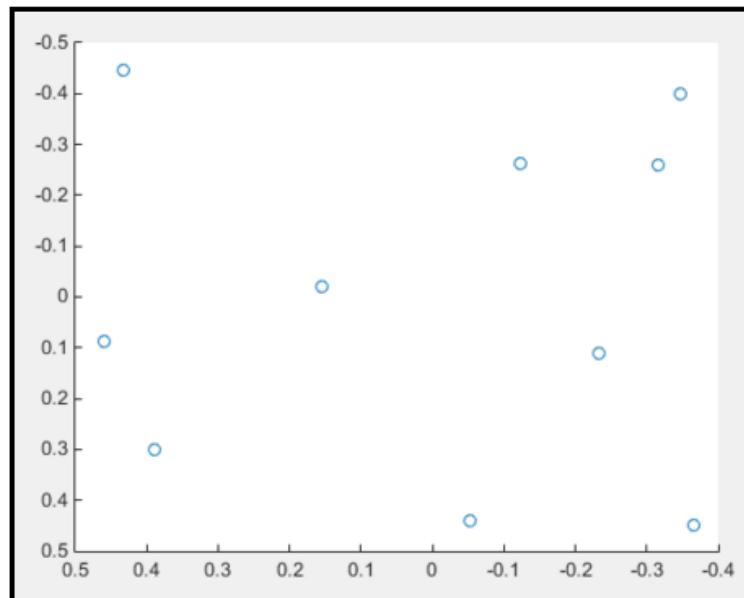


Figure 8.11: Las Ciudades en el plano

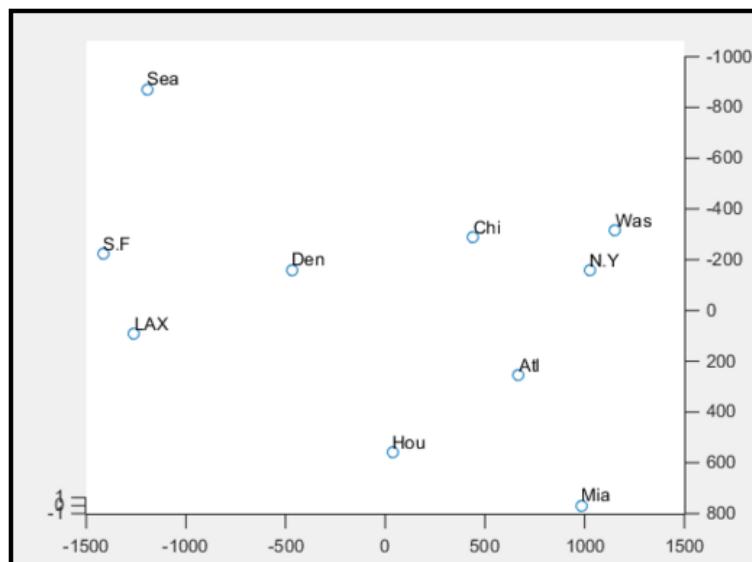


Figure 8.12: Las Ciudades Misteriosas

Aplicando nuestro algoritmo a nuestra matriz de distancias obtenemos la siguiente representación en el plano:

Y con un poco de trabajo podemos llegar a resolver de que ciudades se trata:

Notemos que las coordenadas que nos devuelve MDS pueden estar espejadas y rotadas con respecto a la realidad ya que las distancias en todos los casos son las mismas.

Al utilizar la SVD resolvemos también el problema de la dimensionalidad si es que no conocemos el mismo. Simplemente tenemos que observar la matriz Σ con los valores singulares y ver cuantos valores singulares tiene sentido mantener para conservar la mayor cantidad de energía de la matriz.

8.4.1 Perceptual Mapping

Perceptual Mapping es una aplicación interesante de MDS para aprender los datos a partir de evaluaciones subjetivas u objetivas de los mismos. Supongamos que tenemos un conjunto de autos de diferentes marcas y modelos. Podemos pedirles a los usuarios que nos indiquen que tan parecidos o diferentes son dos autos en un cuestionario o encuesta, con la opinión de muchos usuarios sobre todos los pares posibles podemos construir una matriz de distancias. Luego usando MDS podemos a partir de las distancias construir una representación vectorial para cada modelo de auto en una cierta cantidad de dimensiones que también obtendremos de MDS. Lo interesante es que cada una de estas dimensiones nos va a aportar algún tipo de información sobre los autos, por ejemplo su consumo, seguridad, etc. Podemos ver para cada "feature" cual es el valor que toma cada modelo de auto y deducir cual es el "feature". Lo que estamos haciendo es construyendo los datos mismos a partir de las comparaciones entre pares de datos. Este tema tiene muchas aplicaciones en marketing y estudio de mercado.

8.5 ISOMAP

ISOMAP es nuestro primer algoritmo de Manifold Learning no-lineal, esta basado en principios muy simples y tiene fama de producir resultados por lo menos aceptables en muchos casos. Puede usarse para visualización de datos, como pre-procesamiento o como features a agregar a los datos mismos. El algoritmo consiste en construir un grafo a partir de los datos, calcular las distancias en dicho grafo y luego aplicar MDS para obtener una representación de los datos en pocas dimensiones.

8.5.1 Construcción del Grafo

El primer paso de ISOMAP es muy simple: creamos un grafo no-dirigido en donde cada punto de nuestro set de datos va a estar conectado a sus k vecinos mas cercanos. Para determinar las distancias podemos usar una métrica a elección. En el grafo dos puntos i y j estarán conectados si j está entre los k vecinos mas cercanos de i o si i está entre los k vecinos mas cercanos de j .

Este paso es muy sencillo de completar usando KNN. La matriz del grafo va a tener en cada posición M_{ij} la distancia entre los puntos. El grafo va a ser muy disperso ya que cada punto solo va a tener k vecinos.

8.5.2 Cálculo de Distancias

Una vez que tenemos la matriz de distancias armada vamos a construir una matriz de distancias de todos los puntos contra todos. Para hacer esto hay que aplicar el algoritmo de Floyd-Warshall que dada una matriz de distancias entre pares de puntos nos devuelve la matriz de distancias mínimas de todos contra todos. Floyd-Warshall es $O(n^3)$ por lo que hay que tener un poco de cuidado al aplicar ISOMAP a sets de datos masivos.

8.5.3 MDS

Una vez que tenemos la matriz de distancias de todos los puntos contra todos simplemente aplicamos MDS con $k = 2$ o $k = 3$ para obtener la representación de nuestros puntos en dos o tres dimensiones.

A modo de ejemplo podemos aplicar ISOMAP al set de datos "Swiss Roll" que es un manifold de dos dimensiones dentro de un espacio de tres dimensiones:

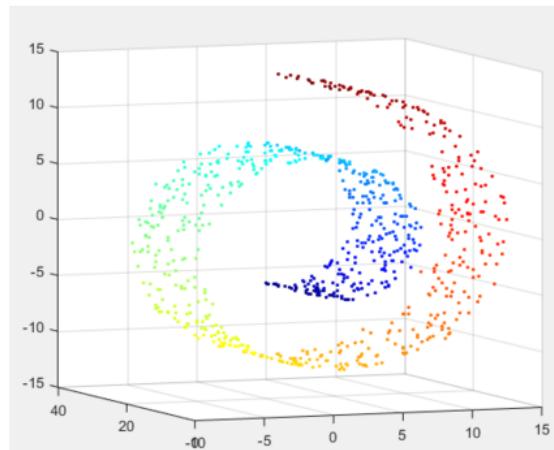


Figure 8.13: Swiss Roll

Como podemos ver ISOMAP hace un buen trabajo en este set de datos distribuyendo los puntos en franjas casi verticales en el plano de acuerdo a su posición en la cinta/rollito/cosito original.

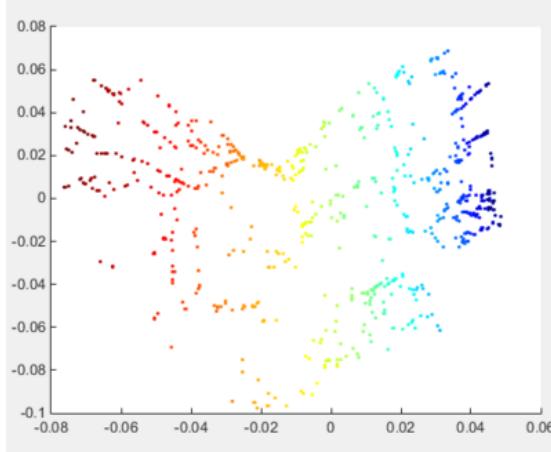


Figure 8.14: Swiss Roll luego de ISOMAP

8.6 Laplacian Eigenmaps

En este algoritmo presentado en [Belkin] vamos a crear un grafo a partir de los datos pero de una forma diferente a la que usamos en ISOMAP, la forma en la cual creamos el grafo aquí es compartida por muchos algoritmos por lo que es muy importante entenderla tanto desde el punto de vista práctico como teórico. La descomposición espectral que vamos a hacer de los datos sirve no solo para este algoritmo sino también para Clustering Espectral, Spectral Hashing y muchos otros algoritmos importantes.

8.6.1 Creación del Grafo

Para la creación del grafo vamos a construir una matriz de adyacencias en donde cada nodo va a estar conectado con sus k vecinos mas cercanos, el peso de las aristas entre nodos vecinos, es decir los valores en la matriz de adyacencias, van a estar dados por la siguiente fórmula:

$$W_{ij} = e^{-\frac{\|x_i - x_j\|^2}{\sigma}} \quad (8.1)$$

El parámetro σ regula el radio alrededor del cual dos puntos se consideran vecinos. Cuando el exponente $\|x_i - x_j\|^2/\sigma$ es muy cercano a cero el valor de W_{ij} es muy cercano a 1 mientras que si el exponente se hace mayor a 4 (aprox) W_{ij} se vuelve 0. Esto nos permite regular cuando consideramos que dos puntos son vecinos y cuando no. Observemos que el tomar los k vecinos mas cercanos es en cierta forma equivalente a establecer una condición de corte para la cual consideramos que W_{ij} es nulo.

La matriz W es una matriz de adyacencias pero no de distancias, cuanto mas grande el valor de W_{ij} mas cercanos son los puntos. En una matriz de adyacencias binaria usamos 1 si los puntos son vecinos y 0 sino, esto es exactamente lo mismo solo que con valores entre 0 y 1 para determinar el grado de vecindad entre dos puntos.

8.6.2 Construcción del Laplaciano

A partir de un grafo y su matriz de adyacencias se puede calcular la matriz Laplaciana del mismo de la forma:

$$L = D - W \quad (8.2)$$

En donde D es una matriz diagonal con el grado de cada vértice del grafo siendo el grado la sumatoria de todas las aristas que inciden sobre el mismo es decir que $D_{ii} = \sum_{j=1}^n W_{ij}$.

La matriz resultante L tendrá en su diagonal el valor del grado ya que la diagonal de W es cero, el resto de los elementos de L es igual a la matriz W cambiada de signo ya que todos los elementos de D que no están en la diagonal son cero. Como consecuencia de esto la suma de cualquier fila y columna de L es cero. Siendo L una matriz simétrica sabemos que todos sus autovalores son reales. El tercer y último paso del algoritmo es descomponer la matriz L en sus autovalores y autovectores.

8.6.3 Descomposición Espectral del Laplaciano

En una matriz Laplaciana el autovalor mas pequeño en valor absoluto siempre va a ser cero, y su multiplicidad va a ser igual a la cantidad de componentes conexos del grafo. Es decir que en un grafo conexo L va a tener exactamente un autovalor igual a 0. Si queremos representar a nuestros datos en k dimensiones debemos tomar entonces los k autovalores mas chicos en valor absoluto ignorando el 0. Los autovectores correspondientes a dichos autovalores son la representación de los datos en k dimensiones.

8.6.4 Teoría de la descomposición espectral

Vamos a intentar demostrar ahora que si los puntos están cercanos en el espacio original entonces la descomposición espectral los mantiene cercanos en el nuevo espacio de dimensiones reducidas.

Vamos a llamar Y_i a los puntos en el nuevo espacio de dimensiones reducidas. Podemos definir entonces la siguiente función costo:

$$J = \sum_{ij} (y_i - y_j)^2 W_{ij} \quad (8.3)$$

Analicemos la validez de esta función. W vale 0 cuando los puntos no son lo suficientemente cercanos en el espacio original, cuando esto pasa permitimos que los puntos estén a cualquier distancia en nuestro nuevo espacio sin costo.

Cuando W_{ij} tiende a 1 queremos que los puntos estén cercanos en el nuevo espacio para minimizar el costo. En resumen queremos minimizar las distancia entre los puntos en en nuevo espacio ponderando por el valor de W_{ij} en el espacio original. Esto tiene mucho sentido.

Theorem 13 (Belkin) *Si los puntos y_i tienen k dimensiones entonces los valores que minimizan J para dichos puntos están dados por la fila i de los k autovectores mas chicos de la matriz Laplaciana.*

Escribimos entonces:

$$\sum_{ij} (y_i - y_j)^2 W_{ij} = \sum_{ij} (y_i^2 + y_j^2 - 2y_i y_j) W_{ij}$$

Sabemos que $D_{ii} = \sum_j W_{ij}$ por lo tanto:

$$= \sum_i y_i^2 D_{ii} + \sum_j y_j^2 D_{jj} - 2 \sum_{ij} y_i y_j W_{ij}$$

Aplicando que $L = D - W$:

$$= 2yLy^t$$

Reemplazando en J :

$$\frac{1}{2}J = \frac{1}{2} \sum_{ij} (y_i - y_j)^2 W_{ij} = yLy^t$$

Por lo tanto podemos escribir:

$$J = \sum_i y_i Ly_i^t$$

Agreguemos la siguiente restricción simplemente para evitar que escalando una solución válida tengamos otra solución válida:

$$y_i Dy_i^t = 1$$

Llamemos λ al valor mínimo de nuestra función costo:

$$\lambda_i = y_i Ly_i^t$$

Como $y_i Dy_i^t = 1$ podemos escribir:

$$y_i Ly_i^t = \lambda y_i Dy_i^t$$

Generalizando:

$$Ly = \lambda Dy \tag{8.4}$$

Resulta ser ahora que la ecuación 6.4 define que y es un autovector de L y λD es un autovalor. Si y es un vector unitario entonces λ es igual a cero ya que la matriz es Laplaciana y la suma de todas sus filas y columnas es cero.

Para evitar que la solución a cualquier problema sea la misma tomamos entonces el segundo autovalor mas chico y su correspondiente autovector y estos serán los que minimicen nuestra función costo. Es decir que si queremos representar los datos en k dimensiones tenemos que tomar los k autovalores mas chicos que sean distintos de cero y sus correspondientes autovectores. Que es lo que queríamos demostrar.

Este teorema justifica desde el punto de vista teórico que la descomposición espectral del Laplaciano que surge del grafo que representa nuestros datos en el espacio original preserva a los puntos que eran cercanos como cercanos en el nuevo espacio sin imponer restricciones sobre los puntos que no eran vecinos. Esta es la base de cualquier algoritmo de Manifold Learning!

8.6.5 Ejemplos

Vamos a ver en primer lugar un ejemplo numérico y teórico para entender el funcionamiento del algoritmo y luego un ejemplo gráfico para ver que resultados genera.

Consideremos los siguientes puntos en dos dimensiones:

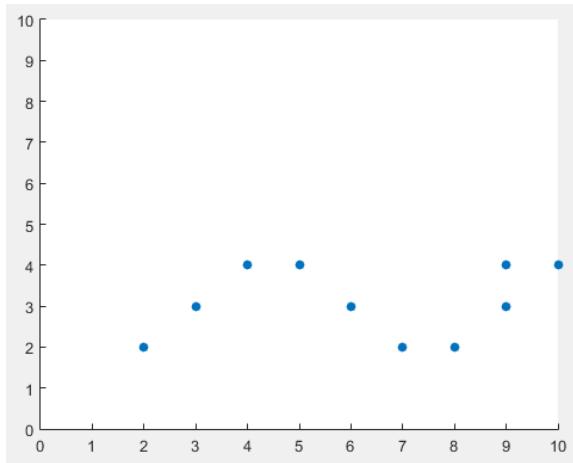


Figure 8.15: Manifold de 1 dimensión en 2D

Esta figura (8.15) es un manifold unidimensional embebido en un espacio de dos dimensiones. Queremos ver si Laplacian Eigenmaps puede descubrir el manifold. Construimos la matriz W usando $\sigma = 2$ y $k = 2$ es decir los dos vecinos mas cercanos. El resultado es la siguiente matriz de adyacencias:

Calculando la matriz D y $L = D - W$ tenemos que L es:

Calculando los autovalores y autovectores de L :

Como era de esperarse el grafo es conexo y por lo tanto L tiene un único autovalor cero. Si queremos representar los datos en una sola dimensión estamos

$$W = \begin{pmatrix} 0 & 0.018316 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.018316 & 0 & 0.018316 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.018316 & 0 & 0.13534 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.13534 & 0 & 0.018316 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.018316 & 0 & 0.018316 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.018316 & 0 & 0.13534 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.13534 & 0 & 0.018316 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.018316 & 0 & 0.13534 & 0.018316 & 0.00033546 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.018316 & 0.13534 & 0.13534 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.00033546 & 0 & 0.018316 & 0 \end{pmatrix}$$

Figure 8.16: W

$$L = \begin{pmatrix} 0.13567 & -0.13534 & -0.00033546 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.13534 & 0.27067 & -0.13534 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.00033546 & -0.13534 & 0.50355 & -0.36788 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.36788 & 0.50321 & -0.13534 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -0.13534 & 0.27067 & -0.13534 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.13534 & 0.50321 & -0.36788 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -0.13534 & 0.50321 & -0.13534 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.13534 & 0.65687 & -0.36788 & -0.13534 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.36788 & 0.73576 & -0.36788 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.13534 & 0.15365 \end{pmatrix}$$

Figure 8.17: L

$$\lambda = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.015549 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.068829 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.13062 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.20745 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.30556 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.39216 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.76608 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.8954 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.924 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.1694 \end{pmatrix}$$

Figure 8.18: Autovalores de L

interesados entonces en el segundo autovector que es:

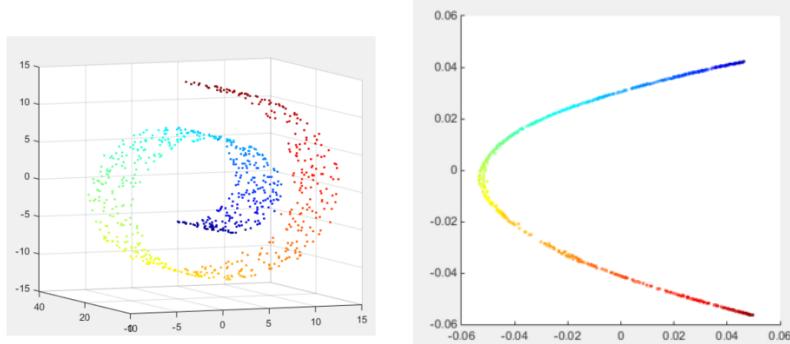
$$V = \begin{pmatrix} 0.459 \\ 0.407 \\ 0.308 \\ 0.258 \\ 0.094 \\ -0.082 \\ -0.143 \\ -0.292 \\ -0.318 \\ -0.33 \\ -0.362 \end{pmatrix}$$

$$V = \begin{pmatrix} -0.302 & 0.459 & -0.494 & 0.438 & 0.126 & -0.337 & 0.364 & -0.004 & 0.025 & 0.011 & 0 \\ -0.302 & 0.407 & -0.244 & 0.017 & -0.066 & 0.422 & -0.69 & 0.019 & -0.14 & -0.065 & 0 \\ -0.302 & 0.308 & 0.13 & -0.42 & -0.157 & 0.228 & 0.256 & -0.065 & 0.62 & 0.301 & -0.002 \\ -0.302 & 0.258 & 0.243 & -0.432 & -0.102 & -0.032 & 0.331 & 0.039 & -0.609 & -0.32 & 0.004 \\ -0.302 & 0.094 & 0.429 & -0.049 & 0.203 & -0.667 & -0.424 & 0.1 & 0.08 & 0.177 & -0.013 \\ -0.302 & -0.082 & 0.397 & 0.381 & 0.197 & 0.204 & 0.049 & -0.405 & 0.242 & -0.537 & 0.08 \\ -0.302 & -0.143 & 0.31 & 0.404 & 0.083 & 0.355 & 0.171 & 0.253 & -0.287 & 0.549 & -0.14 \\ -0.302 & -0.292 & -0.082 & 0.077 & -0.352 & -0.037 & 0.006 & 0.61 & 0.175 & -0.247 & 0.471 \\ -0.302 & -0.318 & -0.155 & 0.004 & -0.41 & -0.129 & -0.043 & -0.057 & 0.065 & -0.137 & -0.756 \\ -0.302 & -0.33 & -0.199 & -0.07 & -0.236 & -0.114 & -0.046 & -0.606 & -0.203 & 0.317 & 0.42 \\ -0.302 & -0.362 & -0.335 & -0.35 & 0.714 & 0.106 & 0.026 & 0.116 & 0.033 & -0.05 & -0.064 \end{pmatrix}$$

Figure 8.19: Autovectores de L

En este autovector los puntos quedan ordenados! Es decir que si los dibujamos sobre una línea nos quedaría P1,P2,P3,P4,P5,P6...P11 que es lo que queríamos pues mantiene el orden de los puntos en nuestro manifold. En una sola dimensión P1 es cercano a P2, P2 es cercano a P1 y P3 etc...

Veamos ahora el resultado de Laplacian Eigenmaps para el Swiss Roll:



Podemos ver un buen resultado, curiosamente podríamos volver a aplicar LE al resultado para obtener una representación en una única dimensión.

8.7 TSNE

T-SNE [van de Maarten] es, al momento de escribir este capítulo, el mejor algoritmo para la representación de datos en dos y tres dimensiones. Es, por lo tanto, el algoritmo más usado y el más importante para visualizar datos multidimensionales en el plano o en 3D.

La idea básica detrás de T-SNE es poder representar datos que originalmente están en muchas dimensiones en solo dos o tres dimensiones de la mejor forma posible. Vimos anteriormente que PCA/SVD lograba esto mismo intentando

preservar la varianza de los datos. Esto es equivalente a que los puntos que están alejados en “n” dimensiones se mantengan alejados cuando los proyectamos en “k” dimensiones.

T-SNE a diferencia de PCA y al igual que en Laplacian Eigenmaps va a intentar que los puntos que estaban cerca en “n” dimensiones se mantengan cercanos en “k” dimensiones. Estas son dos formas radicalmente opuestas de encarar el mismo problema.

Para la formulación de T-SNE vamos a comenzar con una forma de calcular $p_{i|j}$ que es la probabilidad de que j sea vecino dado un punto i .

$$p_{i|j} = \frac{\exp(-||x_i - x_j||^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||^2 / 2\sigma_i^2)} \quad (8.5)$$

El numerador es simplemente un gaussiano alrededor del punto x_i , notemos que cada punto tiene asociado un parámetro σ_i que indica la densidad alrededor del punto para considerar a otro punto como cercano, esta es la diferencia fundamental con otros métodos como Laplacian Eigenmaps en donde el parámetro σ es global, el mismo para todos los puntos. El denominador es la sumatoria de todos los gaussianos para normalizar la probabilidad. Podemos calcular entonces la probabilidad de que dos puntos sean vecinos que es igual a la probabilidad de que i sea vecino de j y j sea vecino de i

$$p_{ij} = \frac{p_{i|j} + p_{j|i}}{2N} \quad (8.6)$$

Llámemos Y a los puntos mapeados a 2 o 3 dimensiones. Calculemos la probabilidad de que dos puntos Y_i y Y_j sean cercanos:

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2) - 1}{\sum_{k \neq i} (1 + ||y_k - y_i||^2) - 1} \quad (8.7)$$

Esto no es un gaussiano sino una distribución T (Student) simplemente porque funciona mejor lo cual se prueba empíricamente.

El objetivo de T-SNE es lograr que si p_{ij} es un valor cercano a 1 entonces q_{ij} sea un valor cercano a 1 y si p_{ij} es un valor cercano a cero entonces q_{ij} puede quedar libre. Esto se logra mediante la divergencia de Kullback-Leibler que estudiamos en el capítulo de teoría de la información y que permite comparar dos distribuciones de probabilidades.

$$KL(P||Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}} \quad (8.8)$$

Y nuestro objetivo es minimizar esta divergencia. Observemos que cuando p_{ij} es muy grande y q_{ij} es chico la divergencia nos da un número muy grande. En cambio cuando p_{ij} es grande y q_{ij} también es grande nos da un número pequeño. Notemos que si p_{ij} es pequeño no importa si q_{ij} es grande o pequeño. Es decir que lo que nos importa es que si los puntos eran cercanos en el espacio original (p_{ij} grande) también sean cercanos en dos o tres dimensiones. Si los

puntos estaban alejados en el espacio original no tenemos preferencias en el espacio reducido.

A modo de ejemplo veamos el resultado de T-SNE para MNIST en donde tenemos datos en 764 dimensiones representando dígitos manuscritos, cada dato es una imagen de 28x28.

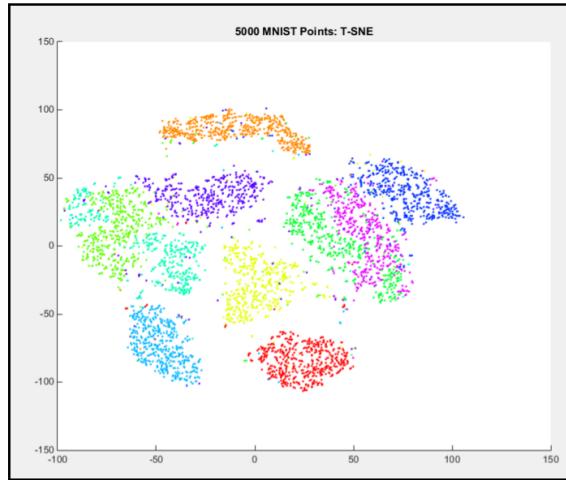


Figure 8.20: T-SNE para MNIST

Como puede verse T-SNE hace un excelente trabajo separando los dígitos en zonas bien diferenciadas en el plano en 2D. T-SNE es una herramienta fundamental en la visualización de datos multidimensionales por lo que siempre resulta conveniente tener una implementación de este algoritmo eficiente disponible en el lenguaje en el que trabajemos.

8.8 Teorema Fundamental de la Dimensionalidad

Uno de los puntos fundamentales cuando usamos algoritmos para reducir la dimensionalidad de un set de datos es saber si es conveniente o no reducir las dimensiones de nuestro set de datos.

Theorem 14 (Teorema Fundamental de la Dimensionalidad) *No siempre es conveniente reducir la dimensionalidad de un set de datos.*

Este teorema se demuestra fácilmente por el **”pigeonhole principle”**. Si siempre fuera conveniente aplicar reducción de dimensiones entonces recursivamente podríamos llevar a cualquier set de datos a una única dimensión. Si aceptamos que esto no tiene sentido entonces el teorema queda demostrado.

La consecuencia directa de este teorema es que debemos analizar en que casos es conveniente aplicar reducción de dimensiones y en que casos no y tenemos también que decidir de que forma hacerlo.

Cuando se trata de visualización T-SNE suele ser un buen algoritmo para empezar, cuando los resultados de T-SNE no son buenos podemos probar con otros algoritmos como ISOMAP, Laplacian Eigenmaps, LLE, Hessian, MDS, Kernel PCA etc. Hay realmente muchísimas opciones.

Si se trata de reducir las dimensiones por cuestiones de performance o para mejorar el rendimiento de un algoritmo la SVD debería ser siempre nuestra primera opción, teniendo en cuenta la posibilidad de usar algoritmos mas avanzados como Autoencoders en caso de ser necesario.

Chapter 9

Information Retrieval

The key utility measure is user happiness. Speed of response and the size of the index are factors in user happiness. It seems reasonable to assume that relevance of results is the most important factor: blindingly fast, useless answers do not make a user happy. However, user perceptions do not always coincide with system designers' notions of quality. For example, user happiness commonly depends very strongly on user interface design issues, including the layout, clarity, and responsiveness of the user interface, which are independent of the quality of the results returned.

- Christopher D. Manning,
*Introduction to Information
Retrieval*

Se conoce como *Information Retrieval* a la rama de la computación que se encarga de recuperar información almacenada en computadoras. En otras palabras hacer consultas sobre información almacenada. La información en general toma la forma de documentos de texto o páginas web, es decir información *no estructurada* para distinguirla de aquella almacenada en una base de datos. Las consultas pueden tomar todo tipo de formas. Vamos a empezar este capítulo hablando de las consultas mas simples que podemos realizar y de las estructuras necesarias para soportarlas, luego a medida que agreguemos consultas mas complejas veremos que estructuras y algoritmos son necesarias para soportarlas.

9.1 Consultas Puntuales y el Dominio de Trabajo

Vamos a definir el dominio de un sistema de information retrieval como un conjunto de **"documentos"** en donde cada documento está compuesto por una secuencia de **términos**, en general un documento será un archivo de texto o una página web, identificados por un *id* mientras que los términos serán palabras. Es decir que buscaremos palabras en textos o páginas web. Si bien esto probablemente cubre el 99% de las aplicaciones de IR debemos notar que podemos redefinir que es un documento y que es un término según nos plazca y todo lo que veremos a continuación puede aplicarse perfectamente en el nuevo dominio.

Una consulta puntual consiste en recuperar todos los documentos que contienen una determinada palabra. Por ejemplo queremos todos los textos que tienen la palabra "Singapur". La aproximación por fuerza bruta consiste en buscar la palabra "Singapur" en cada documento de nuestra colección y luego simplemente devolver los ids de los documentos que matchearon la consulta. Este proceso implica recorrer cada palabra de cada documento en la colección por lo que solo sirve para colecciones realmente muy pequeñas. Para colecciones de un tamaño no mediano a grande necesitamos alguna estructura de tipo índice que nos permita resolver la consulta sin recorrer todos los documentos. La estructura más popular para este tipo de consultas son los llamados *índices invertidos*

9.2 Índices Invertidos

Un índice invertido (II) contiene por cada palabra una lista de los documentos en los cuales aparece dicha palabra. Por ejemplo sea la siguiente colección de tres documentos y su correspondiente II:

```
D1: La historia de Singapur
D2: El precio de la historia
D3: Turismo en Singapur
```

```
de:1,2
el: 2
en: 3
historia: 1,2
la: 1,2
precio: 2
singapur:1,3
turismo: 3
```

Observemos que hemos almacenado los términos ordenados alfabéticamente. Para realizar una consulta puntual como por ejemplo "Singapur" podemos hacer

una búsqueda binaria del término Singapur y luego devolver la lista de documentos en donde se encuentra el mismo : 1 y 3. Para que esto funcione necesitamos explicar que estructuras vamos a manejar en memoria y de que forma construir el índice.

9.2.1 Estructura de un índice Invertido

Todo II se compone de dos partes: por un lado el léxico que es la lista de términos y por otro lado los punteros que son los números de documento en donde ocurren los términos por lo que debemos hablar de como almacenar el léxico, como realizar búsquedas sobre el mismo y luego como almacenar los punteros de forma eficiente.

9.2.2 Almacenamiento del Léxico

El léxico del índice comprende la totalidad de los diferentes términos indexados (no necesariamente todos los términos de la colección). Antes de explicar las diferentes técnicas veamos con que tipo de datos estamos trabajando.



Figure 9.1: Artículo de Reuters

La colección de artículos de Reuters RCV1 tiene un total de 800.000 noticias. Esta colección ocupa aproximadamente 1Gb de texto y tiene un total de aproximadamente 400.000 términos diferentes. La longitud promedio de cada término es de 6 bytes. La colección completa tiene 100 millones de punteros.

Léxico Concatenado

Es decir que el almacenamiento del léxico comprende almacenar 400.000 términos de en promedio 6 bytes. Veamos una primera estructura absolutamente trivial: almacenaremos todos los términos concatenados uno a continuación del otro. Esto implica $400.000 * 6 = 2.4\text{Mb}$. Además de tener a todos los términos concatenados uno a continuación del otro necesitamos alguna estructura en donde podamos buscar. La estructura mas simple es una tabla en donde almacenemos en cada posición el offset en bytes a donde está almacenado el término, como tenemos en total 2400000 bytes para los términos necesitamos $\log_2(2400000) = 22$ bits para los offsets. Tomemos entonces 24 bits que son 3 bytes. La tabla ocupa

entonces $3*400.000 = 1.2\text{Mb}$. Hasta el momento solo hemos usado 3.6Mb de memoria y contamos con una estructura de 400.000 registros de longitud fija 3 bytes en donde podemos hacer una búsqueda binaria. Cuando llegamos a una posición de esta tabla leemos la posición siguiente para saber la longitud del término, luego vamos a la posición indicada, accedemos al término y comparamos contra el que estamos buscando. Nos falta una vez que encontramos el término conectarlo de alguna forma con la lista de documentos en donde aparece, esto lo vamos a hacer mas adelante.

Front Coding

En los casos en los cuales es necesario comprimir mas el léxico podemos recurrir al mecanismo de *front coding*. Este mecanismo aprovecha la particularidad de que en una lista ordenada alfabéticamente de términos los términos vecinos suelen compartir los primeros caracteres. Por ejemplo:

```
advance
advanced
advantage
adventure
advert
advertise
advertisement
advertising
```

Esto puede aprovecharse mediante una estructura en la cual indiquemos la cantidad de caracteres (prefijo) compartidos con el término anterior, la cantidad de caracteres diferentes y un puntero a cuales son esos caracteres.

chars iguales	chars diff	chars
0	8	advance
8	1	d
5	4	tage
3	6	enture
0	6	advert
6	3	ise
9	4	ment
8	3	ing

Veamos algunos detalles: en primer lugar la tercera columna en realidad no tiene cuáles son los caracteres distintos sino un puntero a donde hemos concatenado todos esos caracteres: advancedtagtureadvertisement. Esto es necesario para que la tabla no tenga registros de longitud variable.

En segundo lugar observemos que hemos dividido la tabla en 2 bloques de 4 términos cada uno y que hemos forzado el primer término de cada bloque a

tener 0 caracteres iguales con el anterior, es decir que aunque comparta caracteres igual ponemos 0 lo cual nos obliga a que el puntero apunte al término completo. Esto es necesario para poder realizar una búsqueda binaria del término en cuestión. La tabla funciona ahora como si tuviera n bloques, es decir que la búsqueda binaria va primero al *bloque* del medio, una vez allí como el primer término de cada bloque está completo puede reconstruir todos los términos del bloque y verificar si el término buscado está o no en el bloque.

El front-coding sólo es necesario cuando tenemos poca memoria o bien la memoria que tenemos no es suficiente porque tenemos una enorme cantidad de términos. En general este caso es raro y se suele dar el inverso, el léxico no es un problema en memoria pero queremos poder encontrar los términos muy rápidamente.

Mucha memoria = Mejor Performance

Cuando tenemos memoria suficiente y queremos acceder muy rápidamente a un término podemos prescindir de la búsqueda binaria y usar por ejemplo una tabla de hash para recuperar los términos, si la colección es estática, que frecuentemente es el caso podemos usar hashing perfecto para poder acceder a cada término en $O(1)$. En otros casos podemos usar tablas de hash enormes de forma tal de minimizar las colisiones usando algún método de resolución de colisiones de los que hemos visto. Tengamos en cuenta que en las tablas de hash lo único que hace falta es el puntero al lugar en donde hemos almacenado el término ya sea mediante la concatenación del léxico o front-coding.

Otras estructuras como los tries, tries-compactos o árboles tipo PATRICIA pueden usarse para indexar el léxico pero en general no ofrecen muchas ventajas con respecto a una tabla de hash. En algunos casos la complejidad de estas estructuras hace que una simple búsqueda binaria en una tabla fija funcione mas rápido.

9.2.3 Almacenamiento de Punteros

Hemos visto como almacenar el léxico veamos ahora el caso de los punteros. En el set de datos de Reuters (RCV1) teníamos en total 100 millones de punteros. Veamos cuanto ocuparían los mismos almacenados de forma trivial. Cada puntero es un número de documento, como tenemos 800.000 documentos cada puntero debe ocupar: $\log_2(800000) = 20$ bits, es decir que necesitamos $20 * 100000000 = 250\text{Mb}$. Esto no implica una cantidad astronómica de memoria pero puede ser mucho y para colecciones mas grandes que Reuters podría volverse un problema. Veamos como podemos reducir el espacio necesario para almacenar los punteros.

Observemos que la lista de documentos en donde aparece un cierto término es monótona y creciente, algo del estilo:

elefante: 1,2,3,6,7,10,11

Podemos reducir la entropía de esta lista almacenando las distancias entre los documentos en lugar de los documentos mismos (el primer documento queda igual):

elefante: 1,1,1,3,1,3,1

En esta lista tenemos ahora varias distancias que se repiten mientras que antes teníamos una lista de números diferentes. Sin embargo aun no hemos mejorado ya que la distancia máxima sigue siendo igual al número de documentos porque un término podría aparecer solo en el último documento por lo que seguimos necesitando $\log_2(800000)$ bits por cada puntero. Para salir de esto vamos a proponer almacenar los punteros mediante códigos de longitud variable.

Una forma simple de almacenar números de longitud variable es usando el primer bit de cada byte para indicar si el número continua en el próximo byte. De esta forma los números 0 al 127 los representaríamos en 1 byte con su primer bit en 0, los números 128 al 255 los representaríamos en 2 bytes con un 1 en el primer bit del primer byte y un 0 en el primer bit del segundo byte y así sucesivamente.

Distancia	VBcode
1	00000001
130	1000000100000010
22567	100000011011000010100111

En el caso de Reuters RCV1 usando códigos de longitud variable para los punteros necesitamos solo 116Mb para almacenar los mismos.

Lo único que nos resta es la conexión entre el léxico y los punteros. Los punteros los almacenaremos concatenados uno detrás del otro usando la estructura de números de longitud variable que hemos descripto. En el léxico debemos entonces agregar dos cosas: por un lado un puntero al offset donde comienzan los punteros del término (4 bytes son suficientes) al igual que cuando concatenamos los términos del léxico no hace indicar cuántos punteros tiene cada término porque el offset del término siguiente nos dice donde detenernos.

9.2.4 Estructura General del índice

Estamos en condiciones de explicar la estructura funcional de nuestro índice para resolver consultas puntuales: En primer lugar tendremos todos los términos del léxico concatenados uno detrás del otro y ordenados alfabéticamente. Por otro lado tendremos todas las *distancias* (punteros) concatenadas una detrás de la otra. En ningún caso hacen falta separadores ni longitudes.

Para resolver las consultas booleanas haremos una búsqueda binaria en una tabla que tiene la siguiente estructura:

Term offset (3 bytes)	Pointers Offset (4 bytes)
-----------------------	---------------------------

9.2.5 Almacenamiento de Punteros -Avanzado-

Los códigos de longitud variable son sencillos y en la mayoría de los casos resultan perfectamente aceptables en cuanto al espacio necesario para almacenar los punteros. Sin embargo debemos notar que por lo que sabemos sobre teoría de la información y compresión de datos éstos códigos deben estar lejos de ser el esquema óptimo de compresión para representar los punteros. Hay varias pistas al respecto: la lista de distancias para un término que aparece en muchos documentos estará formada principalmente por distancias pequeñas, por ejemplo si un término aparece en todos los documentos su lista de distancias son todos unos. Sabiendo que las distancias pequeñas son mas frecuentes es un desperdicio usar 1 byte para las mismas y sabemos que nuestros VBcodes son entonces sencillos pero inefficientes.

Antes de atacar el problema tengamos en cuenta que el desarrollo a continuación tiene dos propósitos: por un lado investigaremos el límite al cuál podemos llegar en la compresión de punteros y por otro lado nos da herramientas que son necesarias en aplicaciones en las cuales la cantidad de punteros es realmente astronómica, por ejemplo en un buscador del estilo de Google en donde tenemos miles de millones de documentos.

Código Unario

Los códigos unarios son muy simples: la distancia 1 se escribe 1, la distancia 2 se escribe 01, la distancia 3 se escribe 001, etc. Es decir que escribimos $n-1$ bits en cero y luego un 1. Para la distancia n el código unario ocupa exactamente n bits.

Observemos que los códigos pueden concatenarse uno atrás del otro de forma tal de poder recuperarlos sin necesidad de separadores o de conocer su longitud. Esto es necesario para poder concatenar todos los punteros de todos los términos por lo que buscaremos que todos los códigos que estudiemos cumplan con esta propiedad.

El código unario es muy eficiente cuando tenemos distancias muy pequeñas, ya que solo ocupa 1 bit para la distancia 1 y dos bits para la distancia 2, sin embargo el crecimiento de los códigos unario es muy violento y para la distancia 11000 necesitaríamos 11000 bits que es mucho mas de lo que necesitaríamos si simplemente usáramos VBcodes.

No obstante esto no debemos descartar el código unario ya que es la base de los códigos siguientes y no solo eso sino que en algunos casos es óptimo. Mas sobre esto mas adelante.

Códigos Gamma

Los códigos Gamma se definen como la concatenación de dos binarios:

$$\gamma(x) = \begin{cases} \lfloor \log_2(x) \rfloor + 1, & \text{en unario} \\ x - 2^{\lfloor \log_2(x) \rfloor}, & \text{en binario de } \lfloor \log_2(x) \rfloor \text{ bits} \end{cases} \quad (9.1)$$

Calculemos por ejemplo $\gamma(19)$, $\lfloor \log_2(19) \rfloor = 4$ por lo tanto la primera parte del código es 5 en unario: 00001. Luego necesitamos $19 - 2^4$ en binario de 4 bits es decir 0011. Por lo tanto $\gamma(19) = 000010011$

La decodificación es muy simple: leemos primero la parte unaria que consiste en contar ceros hasta que aparezca el primer 1, el caso de 19 leemos 00001 que es 5 en unario, sabemos que el resto son entonces 4 bits = 3 y el número se calcula como $2^4 + 3 = 19$

Usando códigos delta los punteros de Reuters pueden almacenarse en 116Mb en lugar de 250Mb.

Códigos Delta

Los códigos delta son en general un poco mejores que los gamma ya que el crecimiento de los códigos es un poco mas suave. La definición informal de un código delta es muy simple: un código delta es igual a un código gamma con la salvedad de que la parte que en gamma se escribía en unario en delta se escribe en gamma. La definición formal es:

$$\delta(x) = \begin{cases} \lfloor \log_2(x) \rfloor + 1, & \text{en gamma} \\ x - 2^{\lfloor \log_2(x) \rfloor}, & \text{en binario de } \lfloor \log_2(x) \rfloor \text{ bits} \end{cases} \quad (9.2)$$

La siguiente tabla compara los códigos vistos hasta el momento:

X	VBCode	Unario	Gamma	Delta
1	00000001	1	1	1
2	00000010	01	010	0100
3	00000011	001	011	0101
4	00000100	0001	00100	01100
5	00000101	00001	00101	01101
...
200	1000000101001000	...	000000011001000	000100011001000

Modelo Local de Tipo Bernoulli

Hasta el momento hemos representado a todas las distancias mediante una cierta codificación independientemente del término en el cual aparecía la misma. Si queremos mejorar aun mas el nivel de compresión de las distancias tenemos que analizar a cada término de forma aislada, esto es lo que hace un modelo *local* en contraste con los que vimos hasta el momento que son modelos *globales*

La intuición para los modelos locales es muy simple: en los términos que son muy populares, es decir que aparecen en muchos documentos queremos que

las distancias pequeñas ocupen muy poco pues van a ser mayoría mientras que en los términos más raros no necesitamos que esto pase y no queremos que las distancias muy grandes necesiten codificaciones muy largas.

Necesitamos entonces un sistema de codificación paramétrico y para eso presentamos los llamados códigos de Golomb.

En un código de Golomb vamos a representar a cada distancia en función de un parámetro "b" es decir $\text{Golomb}(X,b)$.

Sea $q = \lfloor \frac{x-1}{b} \rfloor$ y $r = x - qb - 1$

Definimos

$$\text{Golomb}(x, b) = \begin{cases} q + 1, & \text{en unario} \\ r, & \text{en binario en notación prefija} \end{cases} \quad (9.3)$$

r es un resto de dividir por b por lo tanto puede tomar b valores posibles: $0..b-1$, representar los restos en binario de notación prefija significa armar un árbol de Huffman en el cual suponemos a todos los restos equiprobables. Por ejemplo si $b = 3$ tenemos tres restos posibles: (0,1,2) y los podemos representar como 0,10 y 11. Para $b = 6$ los restos serían 00,01,100,101,110,111. Lo importante es que la concatenación de la parte unaria y el resto permiten decodificar el código sin necesidad de separadores o longitudes.

Ejemplo: Sea $b = 3$ y queremos codificar 8 calculamos $q = \lfloor \frac{8-1}{3} \rfloor = 2$ y $r = x - qb - 1 = 8 - 2 * 3 - 1 = 1$ por lo tanto $\text{Golomb}(8,3) = 00110$. Recordando que al resto 1 de dividir por 3 lo habíamos codificado como 10.

Para decodificar primero recuperamos el unario (2) por lo que sabemos que $q = 2$ luego como $b = 3$ usando el árbol de Huffman de los restos de dividir por 3 recuperamos r y el número final es $r + qb + 1 = 1 + 2 * 3 + 1 = 8$. Esto viene de despejar x en la fórmula de r .

El parámetro "b" de Golomb regula la tasa de crecimiento de los códigos, veamos qué pasa si $b = 1$.

$q = x - 1$ y tenemos que escribir $q+1$ en unario lo cual equivale a escribir x en unario y si $b=1$ no hace falta el resto ya que el resto de dividir por 1 siempre es 0, por lo tanto cuando $b = 1$ el código de Golomb es el código unario (!).

Volvamos ahora al modelo Bernoulli. Es hora de explicar el por qué del nombre de este modelo. La suposición es que los términos ocurren o no ocurren en un documento de acuerdo a un proceso de tipo Bernoulli, es decir que un término puede o no ocurrir en cada documento con probabilidad $1/2$ como si dependiera del lanzamiento de una moneda. Esta suposición no es del todo cierta pero sirve a efectos de mejorar la compresión de los punteros.

Suponiendo, entonces, que la aparición de cada término en un documento es un proceso de tipo Bernoulli podemos calcular la probabilidad de que un documento cualquiera tenga un término cualquiera como:

$$p = \frac{f}{N*n} \quad (9.4)$$

Donde: f es la cantidad de punteros (distancias) en nuestra colección. N es la cantidad de documentos y n es la cantidad de términos.

Para Reuters RCV1:

$$p = \frac{100000000}{800000 * 400000} = 0.0003125$$

En el modelo **local** de tipo Bernoulli la probabilidad p debemos calcularla para cada término de la forma:

$$p_i = \frac{ft_i}{N} \quad (9.5)$$

Donde ft_i es la frecuencia del término "i" es decir en cuántos documentos aparece y N es la cantidad total de documentos. p_i en este caso representa la probabilidad de que el término "i" aparezca en algún documento de la colección.

Se demuestra que dada una probabilidad p podemos calcular el b óptimo de Golomb de la forma:

$$b = \left\lceil \frac{\log(2-p)}{-\log(1-p)} \right\rceil \quad (9.6)$$

La siguiente tabla nos muestra el valor óptimo de b de acuerdo a p :

Rango	b
0.3820...0.5000	1
0.2452...0.3819	2
0.1810...0.2451	3
0.1434...0.1809	4

Por ejemplo para Reuters el b global es 2218 (!)

Notemos que p nunca puede ser mayor a $1/2$ ya que si la probabilidad de que un término cualquiera aparezca en un documento cualquiera fuera superior a 0.5 entonces convendría indexar las no-ocurrencias es decir aquellos documentos en los cuales el término no aparece simplemente porque de esa forma almacenaríamos menos punteros. Por ejemplo si tenemos 4 documentos y 3 términos y tenemos 7 punteros sobre 12 posibles es mejor indexar no-ocurrencias y tener 5 punteros.

Vemos también que si $p > 0.3820$ el b óptimo es 1 y sabemos que si $b = 1$ Golomb es el código unario por lo tanto para términos con probabilidad mayor a 0.3820 el código óptimo es el unario. Podemos también probar que unario es una cota superior a lo que pueden ocupar los punteros de un término. Veamos el siguiente ejemplo:

Sea un término que ocurre en los documentos 1,2,4,5,7 y 10. Sus distancias son 1,1,2,1,2,3 y en unario: 1,1,01,1,01,001. Si observamos la concatenación de los códigos unarios: 1101101001 vamos a ver que tenemos en total tantos bits como documentos (10) y que los códigos unarios concatenados forman un bitmap en donde simplemente hay un 1 para los documentos en los cuales el término aparece y un cero en los que no. Claramente nunca los punteros de un término podrán ocupar mas que un bitmap de N bits por lo tanto demostramos

que el código unario es una cota superior a la longitud de los punteros de un cierto término.

Usando el modelo local algunos términos van a tener asociado un b pequeño y otros un b mayor, cuanto más popular el término más chico b , el parámetro b hay que almacenarlo en la tabla que usamos para buscar los términos ya que lo necesitamos para poder decodificar los códigos de Golomb correspondientes a las distancias del mismo.

Veamos ahora que a partir de la probabilidad p podemos calcular la probabilidad de una distancia: la distancia x ocurre cuando el término no aparece en los $x - 1$ documentos anteriores y luego aparece es decir:

$$p(x) = (1 - p)^{x-1} p \quad (9.7)$$

Tomemos el caso de Reuters con $b = 2218$ y veamos cuánto ocuparía la distancia 31278.

$q = \lfloor \frac{31278-1}{2218} \rfloor = 14$ y r ocupa 11 o 12 bits al ser el resto de dividir por 2218. Es decir que el código de 31278 con $b = 2218$ ocupa 26 o 27 bits.

Analicemos cuál es la probabilidad de la distancia 31278:

$$(1 - 0.0003125)^{(31277)} * 0.0003125 = 1.77568212e - 8$$

Y si tenemos la probabilidad de la distancia podemos calcular su longitud ideal si usáramos compresión aritmética:

$$L = -\log_2((1 - 0.0003125)^{(31277)} * 0.0003125) = 26$$

Notablemente es igual a lo que ocupa el código de Golomb. Esto muestra que los códigos de Golomb son una excelente aproximación a la longitud ideal de las distancias de acuerdo a la probabilidad de las mismas modelada como un proceso de tipo Bernoulli. Esta última frase convalida todo lo que hemos hecho hasta ahora con este modelo.

Usando el modelo local de tipo Bernoulli los punteros de Reuters pueden almacenarse en 103Mb, esto no es una gran ventaja con respecto a los códigos Gamma (116Mb) o Delta (111Mb).

9.2.6 Construcción del índice Invertido

Para la construcción del II vamos a seguir un proceso basado en recorrer los datos secuencialmente, generar un archivo auxiliar, ordenarlo y recorriendo el archivo auxiliar ordenado construir el índice invertido.

Usemos el siguiente ejemplo:

- D1: La historia de Singapur
- D2: El precio de la historia
- D3: Turismo en Singapur

Término	Documento
la	1
historia	1
de	1
singapur	1
el	2
precio	2
de	2
la	2
historia	2
turismo	3
en	3
singapur	3

Término	Documento
de	1
de	2
el	2
en	3
la	1
la	2
historia	1
historia	2
precio	2
singapur	1
singapur	3
turismo	3

En la primera fase vamos a recorrer secuencialmente la colección armando un archivo auxiliar en donde indiquemos el término y su documento.

Ordenando el archivo por término y documento.

Recorriendo este archivo secuencialmente podemos construir el índice, el primer término es "de" y ocurre en los documentos 1 y 2, por lo tanto sus distancias son 1 y 1, etc...

9.2.7 Qué Indexar

Parsing

El proceso de extracción de términos a partir de los documentos no es trivial, hay muchas consideraciones a hacer, en primer lugar hay que determinar cuales son los separadores que nos permiten separar los términos, espacios en blanco, puntuación etc. Hay que determinar que hacer con cosas del estilo "Johnson-Johnson" que puede ser uno o dos términos, hay que decidir que hacer con los

números. Por ejemplo no tendría sentido indexar los números de página de un documento pero si cosas como 1984, en concreto si tenemos la frase "2 de Marzo de 1984" tenemos el problema de que hacer con el número "2", si lo indexamos entonces vamos a indexar también números de página, números en tablas y todo tipo de ocurrencias del "2" pero si lo ignoramos no vamos a poder recuperar la frase completa en caso de buscarla. Decisiones...

Case Folding

Una vez tokenizados los términos, proceso que hemos mencionado no es sencillo, hay varias decisiones de pre-procesamiento a tomar. Por ejemplo podemos llevar todos los términos a minúsculas para no tener que indexar "Casa" y "casa" como dos términos diferentes, esto nos puede llevar a que "Azul" y "azul" se consideren lo mismo cuando en realidad el primero puede ser un nombre de persona o de una ciudad.

Stop Words

Es posible también decidir eliminar de nuestro índice las denominadas *stop words* que son palabras muy frecuentes como "a", "que" o en inglés "is", "that", "the", etc. En algunas aplicaciones eliminar stop words es fundamental para el correcto funcionamiento del algoritmo mientras que en otras los stop words ayudan a que el algoritmo funcione mejor. Decisiones.

Stemming

Finalmente podemos decidir aplicar *stemming* que consiste en reducir a las palabras a su raíz idiomática, aplicando stemming cosas como "advertise" y "advertisement" se convierten en un mismo término. Hay varios algoritmos de stemming conocidos siendo el mas popular el algoritmo de Porter. Al igual que en el caso de los stop words algunas aplicaciones se ben beneficiadas por realizar stemming mientras que otras no.

9.3 Consultas de Frases y Proximidad

Nuestro II puede responder, hasta el momento, consultas puntuales de forma eficiente. Esto nos permite también resolver consultas *booleanas* de tipo "t1 and t2" o cosas mas complejas como "(t1 or t2) and (t3)". Para resolver estas consultas simplemente debemos resolver las consultas puntuales correspondientes a los términos involucrados y luego realizar operaciones de unión (or) e intersección (and) entre las listas de documentos que devuelven las consultas puntuales. Es decir que las consultas llamadas booleanas no son mas que operaciones de conjuntos entre consultas puntuales.

Un tipo de consulta importante y que no podemos resolver son las frases o consultas de proximidad como por ejemplo "turismo en París" en donde queremos encontrar los documentos que tienen esa frase. Las frases son una gener-

alización de las consultas por proximidad en donde queremos que los términos de nuestra consulta aparezcan en los documentos cercanos unos a otros, una frase es una consulta con proximidad igual a 1 es decir que cada término debe aparecer a continuación de otro. Si buscamos "torta manzana" con proximidad 2 deberíamos matchear cosas como "torta de manzana", "torta con manzana", "torta sin manzana" (?) etc.

Para resolver estas consultas tenemos que modificar nuestro IR agregando información sobre la posición en la que ocurren las palabras en los documentos.

D1: La historia de Singapur
 D2: El precio de la historia
 D3: Turismo en Singapur

Volviendo a nuestro ejemplo si agregamos la información posicional quedaría algo de la forma:

```
de:1 (3), 2(3)
el: 2 (1)
en: 3 (2)
la: 1(1),2(4)
historia: 1(2),2(5)
precio: 2(2)
singapur: 1(4),3(3)
turismo: 3(1)
```

Los paréntesis contienen la lista de posiciones en donde aparecen los términos, en nuestro ejemplo cada término solo aparece una vez en cada documento pero es solo una casualidad.

Si buscamos la frase "la historia" tenemos entonces que resolver la consulta puntual "la", luego la consulta puntual "historia" y recuperar las listas de posiciones para los documentos en la intersección verificando si existen documentos en los cuales la posición de "historia" es igual a la posición de "la" mas 1.

En cuanto al almacenamiento de la información las posiciones se pueden representar mediante una lista de distancias. Por ejemplo si un término ocurre en el documento 1 en las posiciones (1,3 y 4) y en el documento 3 en las posiciones (2,3,5 y 6) su lista de punteros es algo de la forma: 1,3,1,2,1,2,2,4,2,1,2,1. Analicemos esta lista con cuidado.

1,3,1,2,1,2,2,4,2,1,2,1 indica que el primer documento en donde aparece el término es el 1

1,3,1,2,1,2,2,4,2,1,2,1 es la frecuencia del término en D1 es decir que aparece 3 veces si que leemos los 3 códigos siguientes

1,3,1,2,1,2,2,4,2,1,2,1 significa que el término aparece en D1 en las posiciones 1,3 y 4

1,3,1,2,1,2,2,4,2,1,2,1 el próximo documento es el 3 (2 es la distancia respecto del anterior: 1)

1,3,1,2,1,2,2,4,2,1,2,1 el término aparece 4 veces en el documento 3

1,3,1,2,1,2,2,4,2,1,2,1 las posiciones del término en el documento 3 son 2,3,5 y 6

La información posicional hace que nuestro índice crezca en tamaño notablemente por lo que podría ser conveniente usar códigos gamma o delta para las listas de punteros ya que el tamaño de las mismas podría ser muy grande usando VBcodes.

9.4 Signature Files

En los casos en los cuales un índice invertido no puede usarse por no haber espacio suficiente podemos recurrir a un índice que tiene un costo aun menor en espacio que son los *signature files*. En un signature file vamos a usar funciones de hashing para obtener una representación de cada documento. La representación de cada documento surge de la representación mediante hashing de cada término, a esto lo llamamos el *signature* del término.

Para obtener el *signature* de un término necesitamos k funciones de hashing cuyo espacio de direcciones sea b bits. Por ejemplo si $b = 8$ cada función de hashing devuelve un número entre 0 y 7. El signature consiste simplemente en prender los bits indicados por las funciones de hashing.

Por ejemplo si tenemos el término "hola" y $b = 8$ y decidimos usar dos funciones de hashing tales que $h_1(\text{hola}) = 3$ y $h_2(\text{hola}) = 0$ el signature nos queda $s(\text{hola}) = 10010000$ las colisiones, si ocurren simplemente se ignoran.

Para un documento D el signature del documento es el resultado de realizar un "or" (o raria) de los signatures de todos sus términos.

$$s(D) = \bigvee_{t \in D} s(t) \quad (9.8)$$

Si tenemos el documento $D = \text{"hola mundo"}$ y usamos las mismas funciones que antes y sabemos que $s(\text{mundo}) = 10000010$ entonces $s(D) = 10010010$

9.4.1 Consultas

Para realizar una consulta puntual sobre un signature file partimos de un cierto término "t" al cual le aplicamos las funciones de hashing para obtener su signature, una vez que tenemos el signature tenemos que recuperar todos los documentos para los cuales hay un 1 en las posiciones para las cuales hay un 1 en el signature del término. Estos documentos son los *candidatos* a tener el término. Puede haber falsos positivos ya que en algún documento los bits pueden haber sido encendidos por otros términos diferentes al que estamos buscando (colisiones).

Este método implica analizar el signature del término contra el signature de cada documento, podemos mejorar la eficiencia si en lugar de almacenar la matriz de signatures como documento x bits la almacenamos transpuesta es decir bit x documento. A esto lo llamamos "*bitslices*". Sea por ejemplo $s(t) = 10010001$ para que un documento sea candidato debe tener encendido los bits 0,3 y 7. Es decir que si almacenamos el signature transpuesto podemos

acceder a las filas 0, 3 y 7 y ver cuales son los documentos para los cuales los 3 bits están encendidos. El uso de bitslices implica acceder únicamente a tantos slices como 1s tenga nuestro signature en comparación con acceder a todos los signatures de la colección.

Las consultas booleanas son muy adecuadas para el uso de Signature files, veamos el siguiente ejemplo:

```

Q = (plane and (crash or accident))
s(plane) = 0100
s(crash) = 1000
s(accident) = 1001

```

Para que un documento tenga "crash or accident" debe tener el or de "1000" o "1001". Queremos además que el documento tenga "plane" lo cual quiere decir que el bit 2 debe estar encendido. Por lo tanto estamos buscando documentos que tengan encendido el segundo bit y que tengan ademas encendido el primero o el último bit.

Para que un signature file funcione correctamente la cantidad de 1s y 0s en el mismo debe ser pareja, es decir que tenemos que buscar los valores para b y k la cantidad de funciones de hashing de forma tal que no haya demasiados 1s ni demasiados ceros. Si hay muchos 1s tendremos muchos falsos positivos en las consultas y si hay muchos ceros entonces estamos desperdiando espacio.

9.5 Bitmaps

Un bitmap es una forma de índice muy particular, para construir un bitmap que sirva para un archivo de texto tenemos que tener una entrada por término (igual que en un archivo invertido), pero lo que se guarda para cada término es un vector de N bits (siendo N la cantidad de documentos del texto) donde cada bit es 0 o 1 según se encuentre o no el término en el documento. Por ejemplo tomando cada línea como documento y cada palabra como termino:

```

Pedro y Pablo
Pedro corre
Pablo respira
Pedro corre y respira
Pedro corre Perdo

```

El bitmap correspondiente es:

Los bitmaps son excepcionalmente cómodos cuando se los utiliza para resolver consultas, por ejemplo para las consultas booleanas basta con operar haciendo Ors y ANDs de los bitmaps de los términos buscados para obtener los documentos que son resultado de la consulta. El problema es que son un tanto extravagantes en cuanto al espacio que consumen, para TREC por ejemplo un bitmap ocuparía unos 40 giga-bytes, aproximadamente 20 veces mas que el texto que indexa. Por ello si se usan bitmaps debe usárselos comprimidos

término	D1	D2	D3	D4	D5
pedro	1	1	0	1	1
y	1	0	0	1	0
pablo	1	0	1	0	0
corre	0	1	0	1	1
respira	0	0	1	1	0

9.5.1 Compresión de Bitmaps

Cada entrada en un índice del tipo bitmap tiene la particular característica de estar formada por algunos unos y una enorme cantidad de ceros, para comprimir este tipo de información puede usarse un árbol de derivación binaria.

Supongamos que tenemos la siguiente tira en binario. (64 bits).

```
0000001000000011100000001000000000000000000000000000000000000000000000
```

La técnica de compresión por derivación lo que hace es ir armando un árbol en el cual el bit-vector se procesa en cada paso tomando N bits. N bloques de N bits forman un nodo de N bits en el cual cada bit indica si en el hijo que le corresponde hay o no algún uno. Por ejemplo, si N=4 la primera fase es:

```
0101101000000000
```

```
0000001000000011100000001000000000000000000000000000000000000000000000
```

Una aclaración es que si en cualquier momento la cantidad de bits de un nivel no es múltiplo de N, el último nodo del nivel superior no tendrá N bits. Lo que se debe hacer en este caso es completar dicho nodo con ceros. Hecha la aclaración continuamos comprimiendo el nivel superior del árbol hasta que el nuevo nivel tenga 1 solo nodo, que es justamente lo que ocurrirá:

```
1100
```

```
0101101000000000
```

```
0000001000000011100000001000000000000000000000000000000000000000000000
```

Una vez que se llega a la raíz, lo que se almacena son todos los niveles del árbol uno atrás de otro pero obviando los nodos que son todos ceros. Esto es porque cuando un nodo esta compuesto únicamente por ceros, entonces el bit correspondiente a dicho nodo en el nivel superior será un cero. Por ende, cuando

el descompresor lea ese bit sabrá que ese nodo esta compuesto únicamente de ceros y puede reconstruirlo. Volviendo al ejemplo entonces, lo que se guardará es:

1100, 0101, 1010, 0010, 0011, 1000, 0100

(7x4=28 bits)

Para descomprimir se procede de la siguiente forma: el primer elemento de 4 bits es 1100 lo cual implica que en el nivel inferior los dos hijos de la izquierda tienen algún uno mientras que los dos hijos de la derecha están en cero. Los próximos ocho bits (2 nodos) corresponden entonces a los dos hijos izquierdos.

1100

0101101000000000

Luego se repite lo mismo que antes. Se sabe que el primer grupo de 4 bits es cero, luego hay un grupo que tiene unos que se debe leer y es 0010, luego un grupo de cuatro ceros, luego el 0011 y luego el 1000 luego cuatro ceros, el 0100 y después 9 grupos de cuatro ceros. En el ejemplo se agrupaba un bit cada cuatro, en las aplicaciones reales suele usarse un factor de 8 que comprime mas y es mas fácil de implementar ya que cada byte representa 8 bytes y así sucesivamente.

En este caso comprimimos 64 bits en 28, en general el nivel de compresión de este método es de alrededor del 75%, con lo cual nuestro bitmap gigante pasa a ocupar unos 5 giga-bytes, como puede verse es mucho mas de lo que ocupa el archivo invertido pero el bitmap comprimido puede usarse para realizar consultas sin necesidad de reconstruirlo completamente lo cual resulta muy eficiente. Los bitmaps son estructuras importantes siempre y cuando se manejen de forma comprimida ya que pueden llegar a ocupar muchísimo espacio en su versión en crudo.

9.6 Consultas Ranqueadas

Las consultas ranqueadas son, quizás, las mas comunes en un sistema de IR, se trata simplemente de dada una lista de términos (consulta) obtener una lista de los documentos mas relevantes para la consulta ordenados por relevancia.

9.6.1 BOW y Productos Internos

Una primera aproximación es simplemente contar cuántos términos de la consulta aparecen en cada documento y usar ese puntaje para rankearlo. Esto es equivalente a realizar el producto interno entre el vector consulta y el vector de cada documento usando el modelo de *Bag of Words (BOW)* en donde indicamos con 1 o 0 si el documento tiene o no a cada término de la colección. Por supuesto que en la práctica no hace falta almacenar los vectores ya que podemos resolver la consulta usando un índice invertido. Recuperaremos los documentos para cada

término de la consulta y luego simplemente vemos que documento tiene mayor puntaje, es decir mayor cantidad de términos.

Ejemplo:

```
D1: news about organic food campaign
D2: news of presidential campaign
D3: news of presidential campaign, presidential candidate
D4: news of presidential campaign presidential candidate
D5: news of organic food campaign campaign campaign
```

Si nuestra consulta es "news about presidential campaign" el puntaje de cada documento va a ser 3 ya que en cada documento aparecen 3 (tres) de las palabras de la consulta. Intuitivamente esta no es una buena respuesta a la consulta. Hay dos cosas a observar, en primer lugar "presidential y campaign" deberían tener mas peso que "news" y "about" ya que no estamos buscando "news about" cualquier cosa. En segundo lugar el documento 3 que tiene dos veces la palabra presidential debería tener mas peso que el documento 2 que solo tiene una vez dicho término.

Para solucionar este problema pasaremos al mundialmente famoso modelo TF-IDF.

9.6.2 TF-IDF

Una mejora lógica y automática es contar la cantidad de veces que aparece cada término de la consulta en cada documento. A esto se lo llama *term frequency* y es la parte TF de TF-IDF. Con esta modificación el documento 3 tendría ahora peso 4 en lugar de 3 y sería el primero en nuestra lista de documentos relevantes.

Para solucionar el problema de "about" vs "presidential" recurrimos a IDF. La idea de IDF es muy simple: darle a cada término un peso que sea inversamente proporcional a su frecuencia. Los documentos que aparecen en muchos documentos serán entonces menos importantes que los documentos que solo aparecen en unos pocos documentos.

El IDF de un término se calcula de la forma:

$$\text{IDF}(t_i) = \log\left(\frac{N+1}{ft_i}\right) \quad (9.9)$$

Donde N es la cantidad de documentos y ft_i es la cantidad de documentos en los que aparece el término.

De esta forma por cada término de la consulta y cada documento multiplicamos el valor de TF por IDF y eso nos da el peso del término para ese documento. El puntaje del documento es la sumatoria del puntaje TF-IDF de todos los términos de la consulta que aparecen en el documento.

$$R(Q, D_j) = \sum_{t_i \in Q, t_i \in D_j} ft_{ij} \log\left(\frac{N+1}{ft_i}\right) \quad (9.10)$$

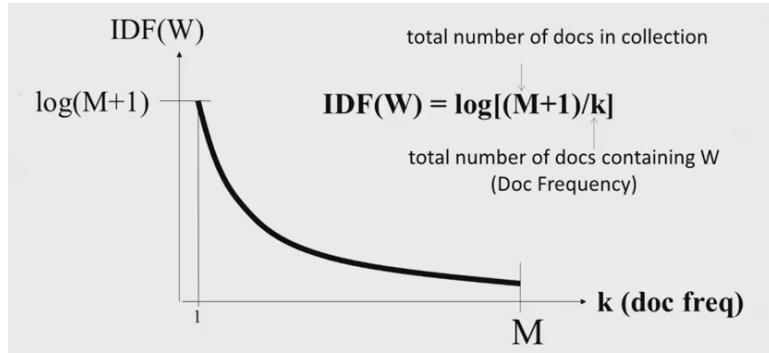


Figure 9.2: IDF

Donde ft_{ij} es la frecuencia del término i en el documento j es decir cuántas veces aparece. N es la cantidad de documentos en la colección y ft_i es la frecuencia del término es decir la cantidad de documentos en las que aparece el mismo.

La figura 9.3 nos muestra un primer ejemplo de TF-IDF en acción. Como podemos ver el documento que tenía "news" y "about" ahora tiene un puntaje mas bajo ya que el IDF de estos términos es pequeño por ser términos populares. El problema es que el documento mas relevante es ahora D5 que no trata sobre "presidential campaign" y esto ocurre porque este documento contiene muchas veces la palabra "campaign". El problema de nuestro esquema es que TF tiene demasiado peso en los puntajes de la consulta, para reducir el peso de TF hay varios esquemas posibles:

1. Usar 1/0 es decir 1 si el término aparece y 0 sino
2. Usar $TF_{ij} = ft_{ij}$ es decir la cantidad de veces que el término aparece en el documento
3. Usar $TF_{ij} = \log(1 + ft_{ij})$
4. Usar $TF_{ij} = \log(1 + \log(1 + ft_{ij}))$

BM25

Una aproximación muy popular y flexible es la definida llamada BM25, usando BM25 el TF de un término se calcula como:

$$TF_{ij} = \frac{(k+1)ft_{ij}}{ft_{ij}+k} \quad (9.11)$$

Cuando $k = 0$ BM25 es el modelo binario es decir que vale 1 si el término aparece en el documento y 0 sino. Cuando $k \rightarrow \infty$ BM25 es el modelo lineal es

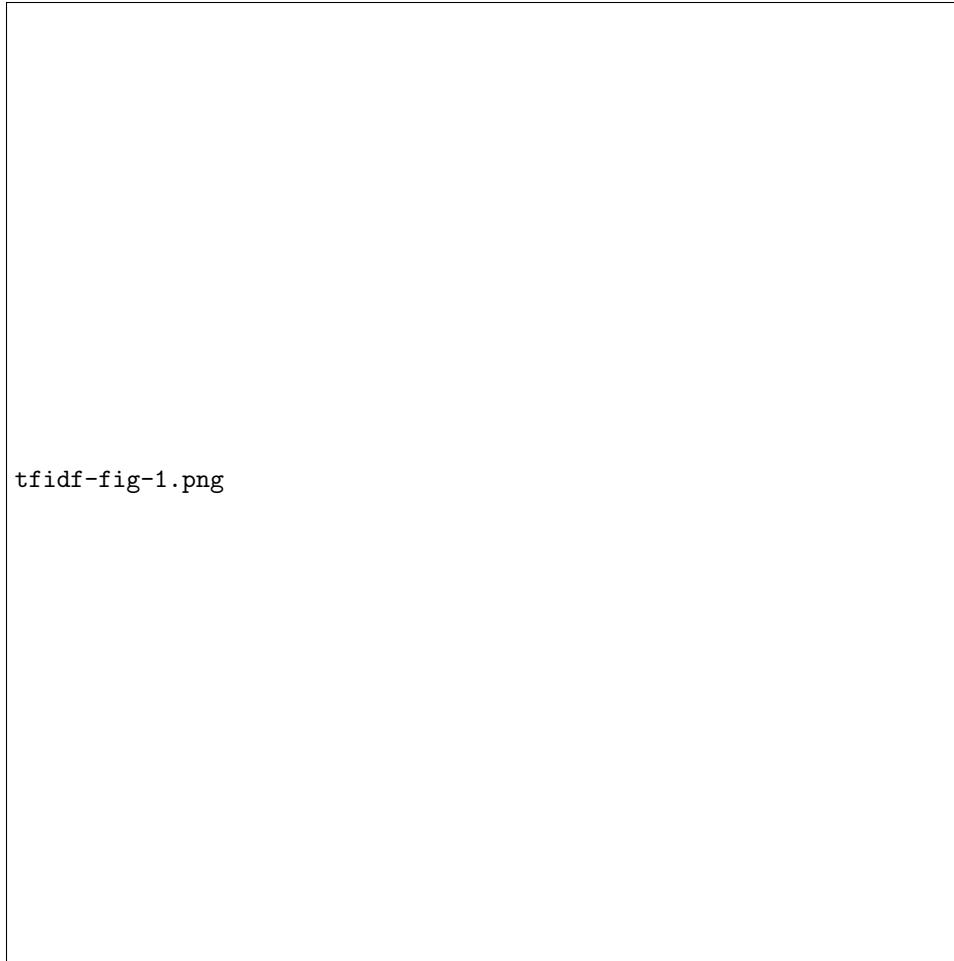


Figure 9.3: Primer Ejemplo de TF-IDF

decir que tiende a tf_{ij} con valores de k intermedios se aproxima a las variantes logarítmicas. Valores de k muy usados en este modelo son $k = 2$ y $k = 1.2$, el motivo es que funcionan bien empíricamente.

La fórmula de ranking con la transformación BM25 queda de la forma:

$$R(Q, D_j) = \sum_{t_i \in Q, t_i \in D_j} \frac{ft_{ij}(k+1)}{ft_{ij} + k} \log\left(\frac{N+1}{ft_i}\right) \quad (9.12)$$

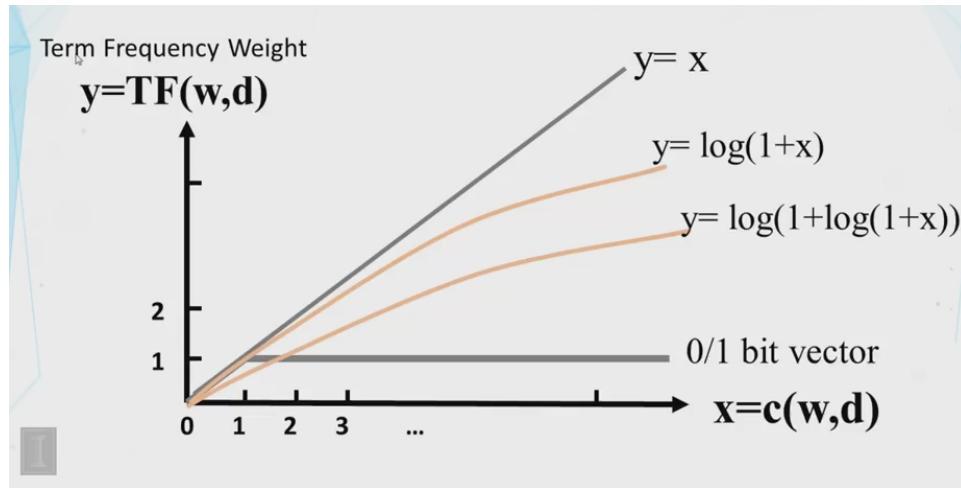


Figure 9.4: TF

Normalización de la Longitud de los documentos

El único problema que nos queda por resolver es que los documentos muy largos tienen ventaja sobre los documentos mas cortos. Un documento que habla "100" veces sobre "presidential campaign" pero que ocupa 1gb probablemente sea menos relevante que un documento que menciona 2 veces "presidential campaign" pero solo tiene dos páginas. El primero podría ser simplemente el dump de una enciclopedia!

La idea es entonces normalizar los documentos de acuerdo a su longitud. Para hacer esto vamos a tomar como pivote la longitud promedio (en palabras) de un documento $avdl$ y vamos a usar la siguiente fórmula para normalizar:

$$\text{norm}_j = 1 - b + b \frac{|D_j|}{avdl} \quad (9.13)$$

Con $b \in [0, 1]$. Cuando $b = 0$ $\text{norm}_j = 1$ y no hay normalización. Cuando $b = 1$ la normalización es $|d|/avdl$ en cuyo caso premiamos a los documentos que son mas chicos que el promedio y penalizamos a los que son mas largos. El valor de b nos da la pendiente o el ratio de penalización. Un valor de b muy popular es 0.75.

Nuestra fórmula final para BM25 es entonces:

$$\text{BM25}(Q, D_j) = \sum_{t_i \in Q, t_i \in D_j} \frac{ft_{ij}(k+1)}{ft_{ij} + k(1-b + b \frac{|D_j|}{avdl})} \log\left(\frac{N+1}{ft_i}\right) \quad (9.14)$$

BM25 y su variante BM25+ que suma una pequeña constante al resultado de

TF_{ij} son dos de las fórmulas mas usadas para ranquear documentos en función de una determinada consulta.

9.7 Indexación Semántica Latente

La indexación semántica latente (LSI por su sigla en inglés) consiste simplemente en aplicar la SVD a una matriz de términos por documentos. La matriz contiene en cada elemento A_{ij} el valor correspondiente al peso del término i en el documento j que se puede calcular mediante TF-IDF o BM25. Una vez que tenemos esta matriz podríamos resolver consultas ranqueadas mediante el llamado método del coseno que consiste en calcular el coseno entre el vector consulta y cada vector documento de la forma:

$$R(Q, D_j) = \frac{QD_j}{|Q||D_j|} \quad (9.15)$$

Si normalizamos los vectores tanto de la consulta como de cada documento entonces el puntaje surge simplemente del producto interno entre los vectores.

Aplicar LSI consiste en descomponer la matriz de documentos x términos aplicando la SVD y quedarnos las k primeras columnas de la matriz U . Al calcular la SVD estamos descomponiendo cada documento en una combinación lineal de elementos de la base V donde cada vector de la base V es una combinación lineal de términos, llamemos a estos vectores, entonces, *conceptos*. Al quedarnos las primeras columnas de U estamos representando cada documento en los k conceptos mas importantes. Esto tiene varios efectos interesantes.

En primer lugar reducimos el ruido de nuestra matriz, en donde antes teníamos una columna por cada término ahora tenemos una cantidad mucho menor de columnas en donde cada columna no es un término sino un concepto. Por ejemplo si un documento era el único que tenía un determinado término en la versión reducida no va ocurrir lo mismo por lo que el ruido de la matriz se reduce. Si buscamos algo de tipo "New England NFL" es probable que documentos que contienen el término "Patriots" sean relevantes aun si no tienen ninguno de los términos de la consulta. Esto se debe a que es muy probable que "New England", "NFL" y "Patriots" formen parte de un mismo concepto(!).

Para poder realizar una consulta es necesario plegar el vector consulta al nuevo espacio de dimensiones, esto lo sabemos hacer e involucra multiplicar al vector consulta por V y Σ^{-1} esto nos da la representación de la consulta Q en k dimensiones que podemos usar para calcular el producto interno contra cada uno de los vectores de los documentos para luego simplemente ordenarlos por puntaje. Una consecuencia favorable es que al ser los vectores mas chicos el proceso es mas eficiente.

Sistemas basados en LSI usando 200 o 300 dimensiones son capaces de responder consultas con calidad comparable a sistemas que usan cientos de miles o millones de dimensiones y por lo tanto son mucho mas lentos. Pero no solo la performance es la ventaja sino que al reducir el ruido en algunos casos los

sistemas basados en LSI pueden resolver consultas mejor que un sistema tradicional. Todo esto nos lleva a tratar de entender cómo evaluar un sistema de consultas.

9.8 Evaluación de Consultas

En esta sección analizaremos de qué forma evaluar el funcionamiento de un sistema de consultas rankeadas o como comparar dos sistemas y decidir cuál es mejor. Cuando comparamos no siempre estaremos hablando de sistemas diferentes sino que puede tratarse del mismo sistema usando diferentes parámetros, es la única forma de realizar la sintonía fina necesaria para encontrar los parámetros que mejor funcionen con nuestro set de datos.

En un sistema de consultas rankeadas la clave es la relevancia de cada documento por lo que será necesaria la intervención humana para determinar cuáles documentos son relevantes y cuáles no de acuerdo a una consulta. Si podemos hacer esto entonces podemos medir la *precisión* y el *recall* de una consulta.

La precisión mide la cantidad de documentos relevantes sobre el total de documentos recuperados. Si nuestro sistema devolvió 10 documentos y 6 son relevantes la precisión es del 60%. El recall mide la cantidad de documentos relevantes recuperados sobre el total de documentos relevantes. Si había un total de 8 documentos relevantes y en los 10 devueltos tenemos 6 el recall es de $6/8 = 0.75$.

	Documentos Recuperados	Documentos no Recuperados
Documentos Relevantes	a	b
Documentos No Relevantes	c	d

$$\text{Precision} = \frac{a}{a+c} \quad (9.16)$$

$$\text{Recall} = \frac{a}{a+b} \quad (9.17)$$

Podemos combinar precisión (P) y recall (R) según un parámetro β de acuerdo a la siguiente fórmula:

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R} \quad (9.18)$$

Cuando $F=1$ se obtiene la medida F_1 que es una de las métricas más populares para evaluar consultas (y otras cosas).

$$F_1 = \frac{2PR}{P+R} \quad (9.19)$$

Recall es una función monótona y creciente de la cantidad de documentos recuperados mientras que la precisión va variando. Consideremos el siguiente ejemplo:

Doc	P	R
D1(r)	1/1	1/10
D2(r)	2/2	2/10
D3	2/3	2/10
D4	2/4	2/10
D5(r)	3/5	3/10
D6	3/6	3/10
D7	3/7	3/10
D8(r)	4/8	4/10
...

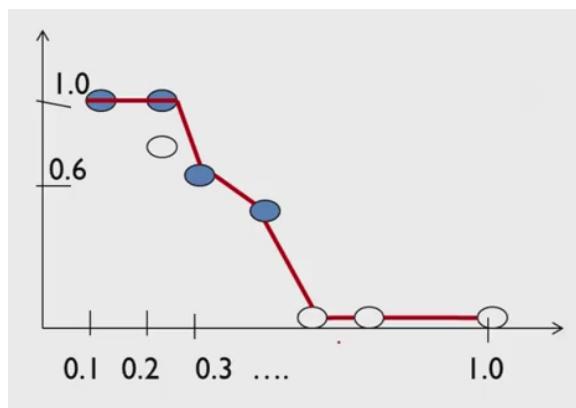


Figure 9.5: Curva de Precisión

La curva de precisión siempre tiene algunos altibajos y finalmente, por supuesto, tiende a cero a medida que necesitamos muchos documentos para capturar la totalidad de documentos relevantes en la lista de documentos devueltos. Una posible métrica para evaluar un sistema de consultas consiste en promediar la precisión para diferentes niveles de recall como por ejemplo 0.1, 0.2, ..., 1.0, a esta métrica se la llama *precisión interpolada*.

Si realizamos una cantidad de consultas podemos hacer un promedio de la precisión interpolada de todas ellas, a esto lo llamamos *mean average precisión (MAP)* el promedio puede ser el promedio aritmético (MAP) o el promedio geométrico (gMAP) recordando(?) que el promedio geométrico es la raíz n-ésima del producto de todos los valores: $(\prod_{i=1}^n x_i)^{1/n}$

Finalmente una forma popular de medir la performance de un sistema de consultas hoy en día es mediante los muy simples, y populares, tests A/B.

9.9 Consultas Probabilísticas

Una aproximación diferente para rankear documentos dada una consulta es el enfoque probabilístico. La idea es muy simple: dada una consulta Q calcular la probabilidad de que cada documento sea relevante para la misma. Una forma de hacer esto es calculando la probabilidad de que el documento sea relevante para cualquier otra consulta, luego la probabilidad es 1 sobre ese número. Lamentablemente esto requiere saber para todas las consultas posibles si cada documento es relevante o no y si sabemos esto probablemente no necesitamos un sistema de consultas⁽¹⁾.

Un giro de tuerca al problema es suponer que cada documento es potencialmente relevante y calcular cuál es la probabilidad de que la consulta haya sido Q

$$P(\text{Relevante}(D_j)|Q) = P(Q|D_j) \quad (9.20)$$

Calculando la probabilidad podemos simplemente ordenar los documentos desde el mas probable al menos probable y esto nos da la relevancia de los mismos. Nuestro problema se resume entonces a calcular la probabilidad de una serie de términos dado un documento tarea de la cual se encarga el área de la computación conocida como *Procesamiento de Lenguaje Natural (NLP)*

9.10 Procesamiento de Lenguaje Natural

El Procesamiento de Lenguaje Natural es una rama de las ciencias de la computación que estudia la interacción entre las computadoras y el lenguaje humano. Abarca una gran cantidad de problemas como spell checking, traducción automática, análisis de sentimiento, responder consultas en lenguaje natural, procesamiento del habla, conversación automática, etc.

Una de las tareas fundamentales del procesamiento del lenguaje natural es la construcción de *modelos de lenguajes*. El objetivo de un modelo es simplemente poder calcular la probabilidad de palabras y frases en un determinado lenguaje. Restringiendo nuestro lenguaje a cada documento tenemos lo que necesitamos para resolver consultas probabilísticas.

9.10.1 Modelo de N-Gramas

El modelo de N-gramas consiste en contabilizar las frecuencias de combinaciones de N palabras. Con N=1 se trata simplemente de contar las frecuencias de las palabras tal y como hicimos antes. Con N=2 contamos las frecuencias de las palabras de acuerdo a la palabra anterior, es decir usando una palabra como contexto. Con N=3 usamos dos palabras y así sucesivamente.

A continuación un fragmento de los bigramas para Alicia en el país de las maravillas.

```
1 "magic": {
    "bottle": 1
```

```

3   },
4     "magpie": {
5       "began": 1
6     },
7     "mail": {
8       "within": 1
9     },
10    "main": {
11      "pg": 1
12    },
13    "maintaining": {
14      "tax": 1
15    },
16    "majesty": {
17      "means": 1,
18      "the": 3,
19      "said": 4,
20      "must": 1,
21      "he": 3
22    },
23  },

```

En el listado anterior $P(\text{must}|\text{majesty}) = 1/11$. De la misma forma podemos calcular cualquier otra probabilidad en base a la palabra anterior.

Y un fragmento de la colección de trigramas

```

2   "glad i've": {
3     "seen": 1
4   },
5   "glad that": {
6     "it": 1
7   },
8   "glad they": {
9     "don't": 1
10  },
11  "glad to": {
12    "do": 1,
13    "find": 2
14  },

```

Es importante destacar como a medida que aumentamos N cada contexto tiene cada vez menos palabras que han ocurrido dentro del mismo. En contextos de 4 o 5 palabras la mayoría de los mismos tendría solo 1 palabra que ha ocurrido.

Otro punto extremadamente importante es que el modelo de N-Gramas siempre incluye a todos modelos de orden menor a N. Es decir que en un modelo de trigramas guardaremos los trigramas, bigramas y unigramas.

A medida que aumenta N la cantidad de espacio necesario para recopilar la información del modelo aumenta. En colecciones de textos grandes es necesario usar una base de datos o un cluster para persistir el modelo ya que el mismo no puede ser almacenado en memoria.

9.10.2 Calculando Probabilidades en el modelo de N-Gramas

Si tenemos una frase como por ejemplo "Alice remained looking thoughtfully" y queremos calcular su probabilidad podemos hacerlo usando la regla de la cadena de la siguiente forma:

$$P(\text{"Alice remained looking thoughtfully"}) = P(\text{Alice}) * P(\text{remained}|\text{Alice}) * P(\text{looking}|\text{Alice remained}) * P(\text{thoughtfully}|\text{Alice remained looking})$$

Aquí podemos notar que para frases largas es, aparentemente, necesario contar con modelos de N-gramas donde N puede llegar a ser un número grande. Esto es costoso y muchas veces inviable ya que frecuentemente solo podemos y queremos almacenar N-gramas hasta un cierto N máximo que hemos prefijado con anterioridad.

La probabilidad de una frase larga puede calcularse con cualquier modelo de N-gramas si asumimos que las probabilidades pueden representarse como un modelo markoviano en donde la probabilidad de una palabra solo depende de las N palabras anteriores. Por ejemplo si queremos la probabilidad de una frase de 5 palabras usando trigramas podemos calcularla de la forma:

$$P(w_1, w_2, w_3, w_4, w_5) = P(w_1)*P(w_2|w_1)*P(w_3|w_1w_2)*P(w_4|w_2w_3)*P(w_5|w_3w_4)$$

Aplicandolo a nuestro ejemplo: $P(\text{"Alice remained looking thoughtfully"})$

$$\begin{aligned} P(\text{Alice}) &= 391 / 4046 = 0.0096 \\ P(\text{remained}|\text{Alice}) &= 1 / 391 = 0.0025 \\ P(\text{looking}|\text{Alice remained}) &= 1/1 = 1 \\ P(\text{thoughtfully}|\text{remained looking}) &= 1/1 = 1 \end{aligned}$$

$$P(\text{Alice remained looking thoughtfully}) = 0.000024$$

De esta forma con un modelo de N-gramas para un N que hemos fijado podemos calcular la probabilidad de cualquier frase que necesitemos calcular o comparar probabilidades de dos o mas frases o calcular la probabilidad de las palabras en base a un cierto contexto anterior.

9.10.3 Maximum Likelihood (Máxima semejanza)

El modelo que hemos construido usando la presunción de Markov se denomina "modelo de máxima semejanza" es decir que es aquel cuyas probabilidades mejor ajustan a los textos que hemos procesado. Para evaluar el funcionamiento de un modelo se suele procesar un cierto porcentaje de los textos que tenemos en nuestra colección dejando de lado otro porcentaje que usaremos para validar

el modelo. Por ejemplo si tenemos 1000 textos podemos procesar 800 para construir nuestro modelo dejando 200 para validar.

Con los 200 textos que hemos dejado afuera luego calculamos la probabilidad de cada texto usando el modelo construido. Esto nos permite comparar diferentes modelos ya que el mejor sera el que nos de un promedio de probabilidades mas altas para los textos de validación.

Es importante usar textos que no fueron parte del proceso con el cual se construyó el modelo para validarlos.

En general para un modelo de N-gramas cuanto mayor sea N mejor será la semejanza del modelo a los textos procesados pero también será mayor la necesidad de espacio para almacenar el mismo. El ajuste es logarítmico, es decir que hay una gran diferencia entre usar solo unigramas o bigramas pero no hay una gran diferencia entre usar hexagramas o eptagramas. En general es muy común que el modelo para colecciones de texto grandes este basado en N=2, 3 o 4.

Ejemplo de texto al azar generado a partir de los bigramas (N=2) de Alicia en el País de las maravillas:

I 'm afraid I know why , of that I 'm a mouse – I wonder is over ,
please ! How dreadfully puzzled. poison , upsetting all come upon
Bill 's a box of me he seems to the young Crab took up into Alice
thought this curious to break the wind , that she had found it was
very much indeed to remain where she was passing at a minute the
hand again very long tail when one side and leave the twentieth time
, You are waiting by his scaly friend of broken

Ejemplo de texto al azar generado a partir de los tetragramas (N=4) de Alicia en el País de las maravillas:

Down the Rabbit-Hole ALICE was beginning to get very tired of
sitting by her sister on the bank , and of having nothing to do :
once or twice she had peeped into the book her sister was reading ,
but it had no pictures or conversations in it , and what is the use of
a book , thought Alice , shall I never get any older than I am now ?
That 'll be a comfort , one way never to be an old woman but then

El modelo de tetragramas luce exactamente como el texto porque para contextos de 3 palabras hay muy pocas palabras posibles de forma tal que lo que hacemos al generar texto al azar es regenerar el texto mismo que hemos procesado!

Para entender el grado de adaptación que puede lograr un modelo vean lo que un modelo de pentagramas puede lograr a partir del texto de la biblia generando texto al azar:

them on beds and couches , that at the same time when Gog shall
come against the fortress. 5:10 They hate him that rebuketh in the
gate , and said , Go in peace : before the LORD is your strength.

8:11 So the Levites stilled all the people , the priests , and they that minister

9.10.4 El Problema de las Probabilidades Nulas

Una vez construido nuestro modelo del lenguaje podemos usarlo para calcular probabilidades de frases tal y como hemos mostrado pero tenemos un grave problema cuando una de las palabras de la frase no aparece en el contexto de N palabras que estamos usando.

Por ejemplo si queremos $P(\text{"Alice joined in"})$ encontramos que "in" nunca aparece con contexto "Alice Joined" por lo que la probabilidad de la frase sería cero. Intuitivamente esto puede parecer correcto pero no lo es ya que la probabilidad de esa frase no puede ser igual a la probabilidad de por ejemplo $P(\text{"jlaldja jdasljas dlkajslsldaj"})$ en donde ninguna de las palabras aparece en el texto. El problema de las probabilidades cero es el punto mas crítico de los modelos probabilísticos para lenguajes, es necesario encontrarle una solución para que el modelo pueda ser usado efectivamente.

En las siguientes secciones vamos a ver diferentes métodos para resolver el problema de las probabilidades iguales a cero.

9.10.5 Smoothing: Corrección de Laplace

La solución mas simple al problema de las probabilidades iguales a cero es sumar 1 a la frecuencia de cualquier palabra en cualquier contexto. A esto se lo llama "ajuste de Laplace"

Partimos de:

$$P(W, \text{Context}) = \frac{\text{count}(W, \text{context})}{\sum_i \text{count}(W_i, \text{context})}$$

En donde $\text{count}(W, \text{context})$ puede ser cero, entonces:

$$P(W, \text{Context}) = \frac{\text{count}(W, \text{context}) + 1}{\sum_i \text{count}(W_i, \text{context}) + V}$$

En donde nos encontramos con "V" que es la cantidad total de palabras que pueden aparecer en cualquier contexto. Este número V es necesario ya que de lo contrario si solo sumáramos 1 a la frecuencia de las palabras las probabilidades podrían sumar mas de 1 y no serían probabilidades. El modelo de Laplace no funciona muy bien y tiene como gran inconveniente la necesidad de poder calcular V. Debemos notar que para calcular V deberíamos conocer el total de palabras posibles en nuestro lenguaje, aun aquellas que no han aparecido en ningún documento que hemos procesado. No siempre es posible conocer este número.

La corrección de Laplace es el método mas simple para corregir las probabilidades iguales a cero pero no siempre es posible implementarlo y no funciona muy bien por lo que solo debe ser usado en casos de emergencia.

9.10.6 Interpolación y Backoff

Una segunda solución al problema de las probabilidades iguales a cero consiste en intentar buscar la palabra en el contexto inmediato inferior. Es decir que si "in" no aparece nunca con contexto "Alice joined" buscamos "in" con contexto "joined" y finalmente buscamos "in" como unigrama es decir sin contexto alguno.

Para el uso de modelos de orden inferior hay dos estrategias posibles. Una es pasar al modelo inferior cuando una palabra no aparece en el modelo utilizado, a esto se lo denomina "backoff". Otra estrategia es usar siempre todos los modelos y calcular la probabilidad como una interpolación de la probabilidad de la palabra en todos los contextos disponibles.

Para interpolar con trigramas necesitamos una fórmula de tipo:

$$P(W, \text{context}) = \lambda_1 P(W|W_1 W_2) + \lambda_2 P(W|W_1) + \lambda_3 P(W) \quad (9.21)$$

Usamos los trigramas, bigramas y unigramas dándoles a cada uno una ponderación con el parámetro lambda.

La elección de los parámetros lambda debe hacerse probando valores y analizando el resultado del modelo obtenido en el set de validación. Recordando que el set de validación es un conjunto de documentos que no han sido procesados para la construcción del modelo. En general puede correrse un algoritmo del estilo grid-search en el cual probamos diferentes parámetros lambda como por ejemplo 0.1, 0.3, 1, 3. El grid-search no son mas que tres ciclos for anidados en donde cada for prueba los 4 valores posibles para lambda.

Stupid Backoff

El algoritmo de Stupid Backoff creado por Brants en 2007 es un método de backoff muy simple que es usado para colecciones de textos muy grande, como por ejemplo la Web, Wikipedia. Etc. El principio básico de un algoritmo de backoff es cuando una palabra no es encontrada en un contexto pasar al contexto de orden inmediato inferior.

La fórmula de Backoff del algoritmo es muy simple:

$$P(W) = \frac{P(W)}{N} \quad (9.22)$$

$$P(W|N) = \frac{\text{count}(W|N)}{\sum \text{count}(W|N)} \quad (9.23)$$

$$P(W|N-1) = 0.4 * \frac{\text{count}(W|N-1)}{\sum \text{count}(W|N-1)} \quad (9.24)$$

Es decir que se usa un simple peso de 0.4 para ponderar hacia abajo el uso de un modelo de orden inferior. Si partimos de tetragramas entonces luego pasamos

a $0.4 * \text{la probabilidad con trigramas}$ luego a $0.4 * 0.4$ por la probabilidad de bigramas y finalmente a la probabilidad de unigramas que no se pondera.

Es fundamental destacar que el algoritmo de Stupid Backoff no genera probabilidades, la suma total puede llegar a dar mayor que uno, lo que nos da es simplemente un puntaje o score para las frases que estemos analizando.

Los métodos que veremos a continuación tienen como objetivo generar probabilidades válidas sin la necesidad de usar interpolación o backoff.

9.10.7 Good Turing Smoothing

Este algoritmo tiene como objetivo equilibrar las probabilidades de las palabras desconocidas con aquellas ya procesadas por el modelo de forma de seguir generando probabilidades válidas.

Vamos a usar como notación N_c como la cantidad de palabras que tienen frecuencia "c" dentro del modelo de orden N. Por ejemplo usando N=1, es decir, unigramas N_{23} es la cantidad de palabras que tienen frecuencia igual a 23.

La probabilidad de una palabra nueva la vamos a estimar en base a cuantas veces hemos visto una palabra nueva en el contexto es decir en base a N_1 es decir que si W nunca ocurrió en un determinado contexto entonces $P(W|\text{context}) = N_1/N$ siendo N la cantidad de palabras en total en el contexto.

Veamos un ejemplo que no tiene que ver con el procesamiento de lenguaje natural sino con la pesca. Hemos pescado 10 truchas, 4 pejerreyes, 2 mojarras, 1 lenguado, 1 salmon y 1 bagre. Cual es la probabilidad de que nuestra próxima captura sea un salmon? Es fácil pues hemos pescado 1 salmon de 19 pescados por lo tanto $P(\text{salmon}) = 1/19$. Que ocurre ahora si queremos calcular la probabilidad de que nuestra próxima captura sea un dorado? No hemos capturado ningún dorado pero hemos capturado una nueva especie 3 (tres) veces por lo tanto $P(\text{dorado}) = 3/19$. Esta es la intuición detrás del algoritmo de Good Turing Smoothing. Estimar la probabilidad de palabras desconocidas en base a cuantas palabras solo hemos visto una vez.

Tenemos ahora una fórmula que nos permite calcular la probabilidad de una palabra que no hemos visto en el contexto:

$$P(W|\text{context}) = \frac{N_1}{N} \quad (9.25)$$

Es necesario ajustar todas las demás probabilidades para que la suma de todas las probabilidades de cada contexto siga siendo igual a 1.

$$C^* = \frac{(c+1)N_{c+1}}{N_c} \quad (9.26)$$

C^* es la nueva frecuencia. La probabilidad se calcula como este nuevo C dividido la suma de las frecuencias originales en el contexto.

Por ejemplo si retomamos el ejemplo de la pesca la probabilidad de Salmon era originalmente $1/16$ pero ahora debemos ajustarla. C que es la frecuencia original es 1 por lo tanto debemos calcular $C^* = (1+1) * \frac{N_2}{N_1}$ donde N_2 es igual

a 1 pues solo hay un pescado con frecuencia 2 (mojarra) y $N_1 = 3$ por lo tanto $C(\text{Salmon}) = 2 * (1/3) = 1$. Es decir que $P(\text{Salmon}) = 2/3 / 18 = 1/27$.

El problema principal del algoritmo de Good Turing Smoothing es que no todos los N_i están definidos. Por ejemplo la palabra "the" ocurre en nuestro texto 1841 veces pero no hay ninguna palabra que ocurra 1840 veces o 1839 veces. En estos casos lo que hay que hacer es estimar las frecuencias de las frecuencias faltantes usando una distribución logarítmica o similar. Debido a esto es que el algoritmo de Good Turing Smoothing no es muy usado en la práctica aunque haciendo la estimación adecuada de las frecuencias faltantes funciona perfectamente.

Es de destacar que si todas las frecuencias ocurren al menos 1 vez es decir si no hay necesidad alguna de estimación el algoritmo de Good Turing Smoothing es óptimo para lograr que el modelo sea el de máxima semejanza a los textos planteados.

La siguiente tabla muestra los valores de "c" una vez realizado el ajuste usando Good Turing Smoothing

Count C	C*
0	0.000027
1	0.446
2	1.26
3	2.24
4	3.24
5	4.22
6	4.19
7	6.21
8	7.24
9	8.25

9.10.8 Algoritmo de Kneser-Ney

Este algoritmo busca aproximarse a Good Turing Smoothing sin la necesidad de calcular el valor de N_i para las frecuencias faltantes. En la tabla de Good Turing Smoothing observemos como C^* puede aproximarse como C menos una cierta constante, en nuestro caso algo semejante a 0.75. Con excepción de los valores para 0 y 1 para los cuales podemos usar una constante diferente.

Podemos entonces estimar la probabilidad de una palabra usando bigramas de la siguiente forma:

$$P(W|context) = \frac{C(W|context)-d}{\sum C(W|context)} + \lambda_1 P(W) \quad (9.27)$$

"d" es el parámetro de descuento para aproximarnos al good turing smoothing y $P(W)$ es la probabilidad del unigrama. λ_1 es el peso del modelo de unigramas y es otro parámetro a determinar.

El algoritmos de Kneser-Key va a intentar también mejorar la probabilidad de los unigramas ya que como vamos a ver a continuación la probabilidad cruda no es una buena idea.

Veamos un ejemplo clásico, sea la siguiente frase (Shannon): "I can't read without my reading" y queremos determinar cual es la palabra que sigue. Si usamos un modelo en el cual usamos tanto bigramas como unigramas vamos a determinar esta probabilidad a partir del contexto "reading" y a partir de los unigramas, es decir sin contexto. El gran problema es que el modelo de unigramas es muy malo para predecir porque va a tender a predecir las palabras que mas aparecen en el texto aun cuando no tengan ningún sentido para la frase en cuestión, esto es lógico pues los unigramas no usan contexto pero no es algo deseable. En nuestro caso "the" sería mas probable para los unigramas que "glasses" y la frase "I can't read without my reading the" no tiene ningún sentido.

Para corregir este problema el algoritmo de Kneser-Ney plantea la probabilidad de los unigramas mediante el cálculo de la probabilidad de que una palabra sea la continuación de cualquier frase. Para entender este concepto pensemos en lo siguiente. En un texto sobre la ciudad de San Francisco la palabra "Francisco" va a tener una frecuencia muy alta en el modelo de unigramas. Sin embargo esta palabra solo ocurre después de "San" por lo tanto su probabilidad de aparecer como continuación de una frase cualquiera es muy baja. Definimos entonces:

$$P_{continuation}(W) = \frac{c(\text{Palabras que preceden } W)}{N} \quad (9.28)$$

Siendo N la cantidad de palabras en total en el modelo de unigramas.

De esta forma una palabra muy popular en la colección pero que solo ocurre en un cierto contexto tiene una probabilidad muy baja de ser la continuación de una frase cualquiera.

Veamos que efecto tendría aplicar el algoritmo de Kneser-Ney a nuestro texto de Alicia en el país de las Maravillas. Vamos a empezar por hacer el ajuste en modelo de unigramas es decir en donde no hay contexto calculando la probabilidad de cada palabra como la probabilidad de que la palabra sea continuación de una frase cualquiera. Es decir que en lugar de contar la cantidad de veces que aparece cada palabra vamos a contar la cantidad de palabras que preceden a la misma.

Al procesar los unigramas de Alicia nos encontramos la palabra "queen" que aparece 73 veces en el texto pero solo tiene 4 predecesores. Por lo tanto la probabilidad de la palabra "queen" al aplicar Kneser-Ney baja considerablemente. Otras palabras como por ejemplo "course" sufren una variación mucho menor por tener un número de predecesores similar a su frecuencia, en este caso "course" tiene frecuencia 28 y 23 predecesores.

Actualizamos ahora la fórmula de Kneser-Ney para bigramas:

$$P(W|context) = \frac{C(W|context)-d}{\sum C(W|context)} + \lambda_1 P_{continuation}(W) \quad (9.29)$$

En donde vamos a calcular a λ_1 como:

$$\lambda_1 = \frac{d}{\sum C(W|context)} c(words following context) \quad (9.30)$$

Donde $c(words following context)$ es la cantidad de palabras diferentes que siguen al contexto en el cual estamos.

A partir de la fórmula de Kneser-Ney para bigramas podemos generalizar una fórmula recursiva para N-gramas:

$$P(W|context_i) = \frac{c(W, context_i)-d}{\sum c(W, context_i)} + \lambda_{i-1} P(W|context_{i-1}) \quad (9.31)$$

$$P(W|context_0) = P_{continuation}(W) \quad (9.32)$$

$$P(W|context_0) = \frac{c(Palabras que preceden W)}{N} \quad (9.33)$$

$$\lambda_i = \frac{d}{\sum C(W|context_i)} c(words following context) \quad (9.34)$$

El algoritmo de Kneser-Ney es el mas usado y uno de los mas efectivos para la construcción de modelos para un lenguaje. Recordemos que teniendo un modelo podemos usar el rankeo probabilístico para determinar la relevancia de un documento con respecto a una consulta.

9.11 Learning to Rank

Recuperar los documentos que matchean una consulta es una tarea mucho mas sencilla que rankearlos, la mayoría de los search engines modernos *aprenden* a rankear los resultados en lugar de decidirlo de forma independiente. Esto tiene sentido porque los buscadores pueden capturar una enorme cantidad de información cada vez que el usuario hace una consulta. La hipótesis es que los documentos relevantes para una consulta son aquellos que el usuario clickea, esto es bastante lógico. El problema consiste en como rankear los documentos para los cuales no tenemos información. La idea de *Learning to Rank* (LETOR) es entrenar un algoritmo que pueda predecir el ranking de los documentos en base a ciertos *features* en los mismos. El proceso entonces es el siguiente:

1. El usuario formula una consulta

2. Recuperamos los documentos que matchean la consulta (pueden ser muchos), esto puede hacerse mediante un índice invertido por ejemplo.
3. Aplicamos un algoritmo a los features de cada documento recuperado
4. Ordenamos los resultados de acuerdo al puntaje que nos devolvió el algoritmo

Los algoritmos para Learning to Rank se dividen en tres grupos:

- Por puntos (pointwise)
- Por parse (pairwise)
- Por listas (listwise)

9.11.1 Algoritmos Pointwise

El input en estos algoritmos es un documento y un query y el resultado es un puntaje. El puntaje surge de aplicar una combinación lineal de pesos a los features del documento, algunos features populares son:

- El valor de Pagerank del documento
- El resultado del coseno entre el documento y la consulta
- El resultado del algoritmo HITS para el documento
- ¿Aparece alguna palabra del query en el título del documento? (1,0)
- ¿Aparece alguna palabra del query en color en el documento? (1,0)
- ¿La url del documento contiene "ñ"? (1,0)
- Tamaño mínimo de la ventana tal que contenga todas las palabras del query (*window*)
- Cantidad de imágenes en el documento
- Longitud de la URL
- Longitud del documento
- Lenguaje del documento
- Fecha de última modificación del documento
- Cantidad de links en la página
- Cantidad de veces que el documento fue clickeado para la consulta
- Cantidad de veces que el usuario visitó el documento (feature personalizado)

En Google se usan mas de 200 features para rankear los documentos.

Los features idealmente deben estar normalizados para que no haya grandes diferencias de escala, esto sirve para que el algoritmo aprenda mas rápido la importancia de cada feature.

example	docID	query	cosine score	ω	judgment
Φ_1	37	linux operating system	0.032	3	relevant
Φ_2	37	penguin logo	0.02	4	nonrelevant
Φ_3	238	operating system	0.043	2	relevant
Φ_4	238	runtime environment	0.004	2	nonrelevant
Φ_5	1741	kernel layer	0.022	3	relevant
Φ_6	2094	device driver	0.03	2	relevant
Φ_7	3191	device driver	0.027	5	nonrelevant

Figure 9.6: Learning to Rank

Un simple clasificador lineal es frecuentemente suficiente para rankear los documentos de forma efectiva. Comenzamos con un vector w de pesos al azar para cada feature. El ranking de un documento es simplemente el producto interno entre el vector de pesos y el vector de features del documento.

$$\text{Rank}(D) = w * f_d \quad (9.35)$$

El objetivo es encontrar w , podemos usar como set de entrenamiento un conjunto de documentos para los cuáles tenemos una cierta cantidad de clicks.

Sea el siguiente ejemplo:

DocId	Coseno	PageRank	window	lastmodif	clicks
331	0.7653	0.023	4	5	3
311	0.8182	0.015	2	12	1
19	0.6671	0.013	2	3	31
993	0.8131	0.018	4	2	11
991	0.7761	0.013	3	9	5

Queremos encontrar w que minimice la diferencia de cuadrados con el vector de clicks es decir que $w * f_i$ de cada documento nos aproxime a la cantidad de clicks que tiene el mismo. De esta forma dados los features de un documento podemos predecir su cantidad de clicks sin necesidad de haberlos registrado. Los clicks sirven únicamente para entrenar el algoritmo!

El algoritmo para encontrar w es Gradient Descent y es muy usado para resolver todo tipo de problemas de optimización ya que escala muy bien a datos masivos.

Cada vez que el usuario hace click en una página actualizamos:

$$w = w - \alpha(w * x - y) * x \quad (9.36)$$

En donde x es el vector de features del documento y y es la cantidad de clicks reales. Luego de procesar millones de queries el algoritmo aprende los pesos necesarios para w y podemos rankear en base a $w * x$ y continuar actualizando w

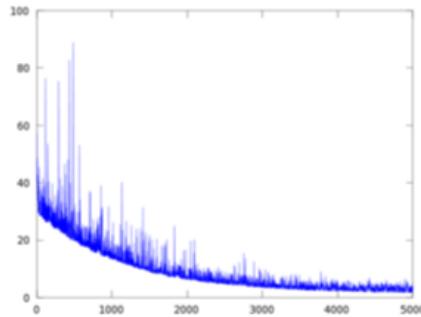


Figure 9.7: Error de Gradient Descent

En nuestro ejemplo obtenemos $w = (28, 14, 0.51, -6.1, -2.13)$, (28 es el término independiente) si multiplicamos este w por los vectores de cada documento obtenemos $(4, 2, 19, 12, 2)$ y la cantidad de clicks reales era $(3, 1, 21, 11, 5)$ como podemos ver es una aproximación muy buena. El vector w nos da una interpretación de la importancia de cada feature. El coseno es el feature mas importante (14), seguido del pagerank (0.51), luego viene el tamaño de la ventana en donde podemos encontrar las palabras del query. Este peso es negativo, cuanto mas grande la ventana peor rankea el documento lo cual es absolutamente entendible. La fecha de última modificación, que es un offset, tiene peso negativo. Esto tiene sentido ya que a mayor valor en dicho feature mas viejo es el documento, aparentemente los documentos mas actuales rankean mejor que los mas viejos.

9.11.2 Algoritmos Pairwise y Listwise

En eset caso entrenamos en base a pares de features y el algoritmo solo tiene que decir cuál de los dos documentos es mas relevante, es decir que entrenamos una función de comparación. El resultado es +1 si el primer documento es mas relevante o -1 si el segundo es mas relevante.

Con la función de comparación que aprendemos podemos luego ordenar los documentos.

Es posible convertir un algoritmo pairwise en un algoritmo pointwise, solo tenemos que concatenar los vectores de dos documentos y el valor target y es 1 o -1 según cual tenga mayor cantidad de clicks.

Finalmente los algoritmos de tipo *listwise* procesan una lista de documentos y devuelven la lista ordenada. Son mas raros que las otras dos variantes.

Existen algoritmos específicos para rankear como por ejemplo SVMs que veremos mas adelante.

Chapter 10

Visualización

The minimum we should hope for with any display technology is that it should do no harm.

- Edward Tufte

En este capítulo abordamos el tema de visualizar datos, las visualizaciones son fundamentales por dos motivos: en primer lugar para entender los datos desde el punto de vista del análisis exploratorio y también para comunicar resultados como parte final del proceso de Data Science. La forma en que visualizamos en un caso u otro es completamente diferente.

Cuando visualizamos como parte del análisis exploratorio de los datos creamos muchas visualizaciones simples sin preocuparnos demasiado por el aspecto estético de las mismas, este tipo de visualizaciones son meramente funcionales y tienen como objetivo descubrir información acerca de los datos. Mediante plots podemos entender cómo se distribuyen las variables, la relación que existe entre diferentes variables de nuestro set de datos, cuáles atributos sirven y cuáles no para la pregunta que queremos responder, podemos detectar datos anómalos, que pueden indicar errores en el proceso de obtener y depurar los datos. Podemos observar como se agrupan los datos en clusters si los representamos en 2D. En el análisis exploratorio usamos un conjunto de plots standard como herramientas que aplicamos a los datos para entenderlos.

Las visualizaciones que tienen como objetivo comunicar los resultados del proceso de Data Science son menos y el aspecto estético de las mismas es mucho mas importante. En estas visualizaciones es normal crear plots ad-hoc que sean específicos para lo que queremos comunicar o sean variantes de plots conocidos. Es fundamental que la visualización transmita la historia que queremos contar, que el público pueda entenderla y obtener a partir de ellas las conclusiones que obtuvimos en el proceso de data science que aplicamos a los datos.

En este capítulo vamos a comentar algunas reglas básicas y fundamentales sobre visualización de datos y luego vamos a presentar los tipos de plots mas

comunes comentando acerca de como usarlos. En algunos casos usaremos *ggplot2* en R para crear los plots que mostramos.

10.1 Principios Básicos

10.1.1 No usar tres dimensiones de forma injustificada

Todas las visualizaciones se presentan en papel o pantallas que tienen dos dimensiones, el uso de plots de 3-d que necesariamente deben proyectarse en 2-d está desaconsejado salvo que sea absolutamente necesario. Por ejemplo el uso de plots en 3-d puede ser necesario si lo que importa es la forma que tienen ciertos objetos en el espacio tridimensional.

En los plots en 3-d la profundidad puede producir efectos no deseados, es muy común encontrar plots tridimensionales en los cuales ciertas áreas parecen mayores que otras que son iguales simplemente por la sensación de estar mas cerca del lector.

10.1.2 El uso del plano

La posición de los objetos en el plano es una de las fuentes de información mas importantes en un plot. En general la información en el eje *Y* tiene mas relevancia que la información en el eje *X*, esto probablemente se deba a que asociamos el eje *Y* con el efecto de la gravedad. Es por esto que en gráficos que muestran la evolución de una variable a lo largo del tiempo el tiempo debe ir en el eje *X* y la variable de interés en el eje *Y*.

10.1.3 Oclusión de Información

La pauta de profundidad mas importante que podemos recibir es la oclusión, es decir cuando un objeto o parte de un objeto queda oculta detrás de otro. La información visual es que el objeto parcialmente oculto está detrás del objeto que podemos ver completo. En líneas generales esto implica que el objeto en primer plano es mas importante.

La oclusión puede usarse en plots para relativizar la información de los objetos poniendo a aquellos que el lector debe visualizar mas tarde en un segundo plano.

10.1.4 Mal uso del Plano

Así como no debemos usar visualizaciones en 3-d cuando dos dimensiones son suficientes tampoco es recomendable usar plots en dos dimensiones cuando una simple lista en una dimensión es suficiente.

Las listas son muy efectivas cuando el orden de los objetos en la visualización es importante, por ejemplo ordenar textos en forma alfabética ayuda a poder

encontrar rápidamente un texto o etiqueta que nos interese, en contraste encontrar un nodo por su etiqueta en 2-d puede ser una tarea costosa que requiere que el usuario busque por todo el espacio de la visualización

10.1.5 Capacidad de Retención

En general la regla de 7+/-2 es una buena aproximación a la cantidad de información que el cerebro puede retener a muy corto plazo, cuando creamos visualizaciones debemos tener en cuenta esta regla para no crear una cantidad excesiva de elementos visuales que dificulten el proceso de comprensión de las mismas.

10.1.6 Ceguera al cambio

En visualizaciones que buscan comparar diferentes cosas mediante varios plots debemos tener en cuenta que en todas las personas la capacidad de detectar cambios es muy pobre cuando se centra la atención en otro objeto. Es decir que si queremos que el usuario haga una comparación debemos asegurarnos que el centro de atención de nuestra visualización esté precisamente en el objeto que cambia.

10.1.7 Uso del Color

Si bien el uso del color ayuda en muchas visualizaciones es un buen principio asegurarse que las mismas puedan entenderse correctamente en blanco y negro, esto tiene varias ventajas: en primer lugar nos aseguramos que las personas con problemas de daltonismo puedan entender la visualización correctamente, en segundo lugar nos aseguramos que nuestra visualización pueda verse correctamente en un e-reader o luego de imprimirse en B&N. Finalmente si la visualización es efectiva en blanco y negro entonces podemos estar seguros de que el color es un elemento que ayuda a la misma y no parte de la visualización.

En la temporada 2015 la NFL estrenó uniformes nuevos para el partido entre los Jets de NY y los Bills de Buffalo, los uniformes de fuertes tonos rojos y verdes eran fáciles de distinguir en color pero lamentablemente resultaban prácticamente idénticos bajo una de las formas mas comunes de daltonismo.

Listing 10.1: .

Que una organización tan importante haya caído en un error tan crítico resulta imperdonable pero sirve como ejemplo para recordar que nuestras paletas de colores siempre deben chequearse de forma tal que sean distinguibles para las personas con daltonismo.

10.1.8 Foco en la Funcionalidad

La funcionalidad de una visualización siempre debe ser la prioridad número uno. Una visualización funcional pero estéticamente desagradable puede ar-



Figure 10.1: Jets vs Buffalo

reglarse mediante la aplicación de principios de diseño gráfico. Una visualización estéticamente atractiva pero que no funciona es muy difícil de corregir y a menudo el tiempo invertido en la parte cosmética de la misma se pierde por tener que rediseñar toda la visualización.

10.2 Principios de Visualización de Tufte

Edward Tufte es una de las personas con mayor influencia en el mundo de la visualización de datos siendo el autor de los libros "The Visual Display of Quantitative Information" y "Envisioning Information" los cuales dieron origen al estudio de la visualización de datos como un área importante dentro de Data Science. En esta sección resumimos algunos de los puntos claves de los principios de Tufte sobre visualización de datos.

10.2.1 Excelencia

Sobre la excelencia en la visualización de datos:

- Consiste en comunicar ideas complejas con claridad, precisión y eficiencia.
- Es lo que le da al público la mayor cantidad de ideas en la menor cantidad de tiempo usando la menor cantidad de tinta posible
- Requiere decir la verdad sobre los datos.

Las visualizaciones deben:

- Mostrar los datos
- Inducir al público a pensar sobre la sustancia en lugar de la metodología, diseño, tecnología o cualquier otra distracción
- Evitar distorsionar lo que los datos dicen
- Presentar mucha información en un espacio reducido
- Permitir entender sets de datos masivos
- Estimular el ojo a comparar diferentes datos
- Revela los datos en diferentes niveles de detalle
- Servir un propósito claro

10.2.2 Principios

Las visualizaciones deben orientarse a los siguientes objetivos:

- Focalizar en el contenido
- Comparación en lugar de descripción
- Integridad
- Alta resolución
- Uso de diseños y conceptos clásicos

Foco en el contenido: Por sobre todo lo más importante es mostrar los datos. El foco debe estar en el contenido de los datos y no en la técnica de visualización empleada. Esto nos lleva a que el diseño debe ser transparente. El éxito de una visualización está basado en el conocimiento de la sustancia que los datos nos transmiten y la calidad, relevancia e integridad del diseño empleado para mostrar esta sustancia. Hay que asumir que el público es tan inteligente como la persona que crea la visualización. Nunca hay que simplificar una visualización asumiendo que el público no va a entenderla.

10.3 Scatter Plots

Un **scatter plot** es una de las visualizaciones más comunes y versátiles, principalmente porque puede adaptarse para diferentes tipos de datos y permite acumular un buen número de dimensiones en un mismo plot.

En principio en un scatter plot representamos dos variables numéricas en los ejes X e Y y por cada instancia de nuestro set de datos dibujamos un punto en las coordenadas indicadas.

Estos plots nos dan una idea de la dependencia que existe entre las dos variables y de las características de esta dependencia: lineal, no-lineal, etc.

A un scatter plot básico se le pueden agregar dimensiones controlando el color de los puntos, como en el siguiente ejemplo para el conocido set de datos IRIS.

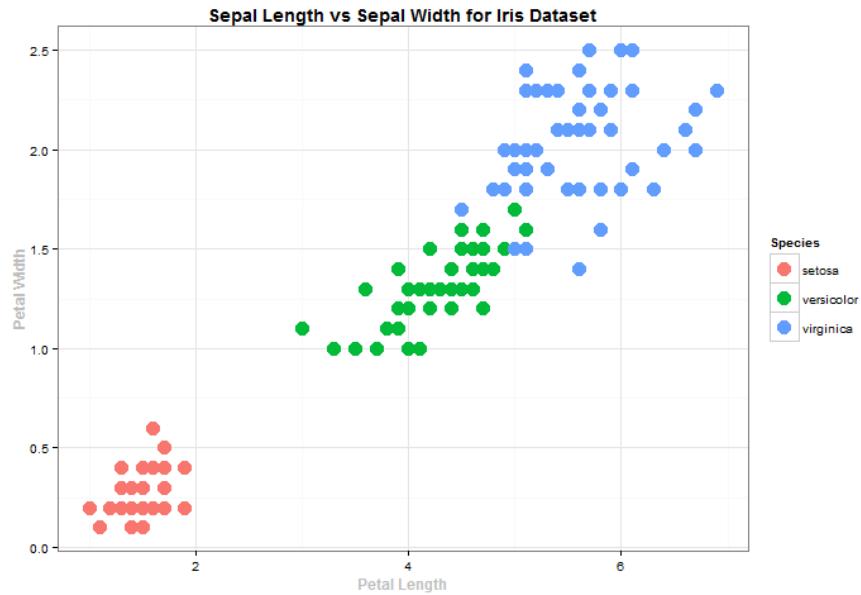


Figure 10.2: Scatter Plot

```
1 ggplot(data=iris, aes(x=Sepal.Length, y=Sepal.Width, color=Species))+
2   geom_point(size=5)+  
3   xlab("Sepal Length") +  
4   ylab("Sepal Width") +  
5   ggtitle("Sepal Length vs Sepal Width for Iris Dataset")
```

Listing 10.2: Scatter Plot Para Iris

En el plot vemos la relación entre el largo y ancho de los Pétalos en IRIS mientras que el color muestra la especie recolectada. Claramente hay una fuerte relación lineal positiva entre el largo y ancho de los pétalos. También podemos ver que la especie IRIS setosa tiene pétalos más pequeños que las otras dos especies y que el tamaño de estos debería ser suficiente para discriminar esta especie de flor de las otras dos. Entre IRIS versicolor e IRIS virginica el tamaño de los pétalos también es un buen discriminador pero hay una cierta superposición que puede generar algunos errores de clasificación si nos basamos únicamente en esta variable.

El color puede representar una variable tanto categórica como numérica en cuyo caso será un gradiente de color.

Es posible cambiar la forma de los puntos de acuerdo a una variable categórica: círculos, cuadrados, estrellas, triángulos, rombos, etc.

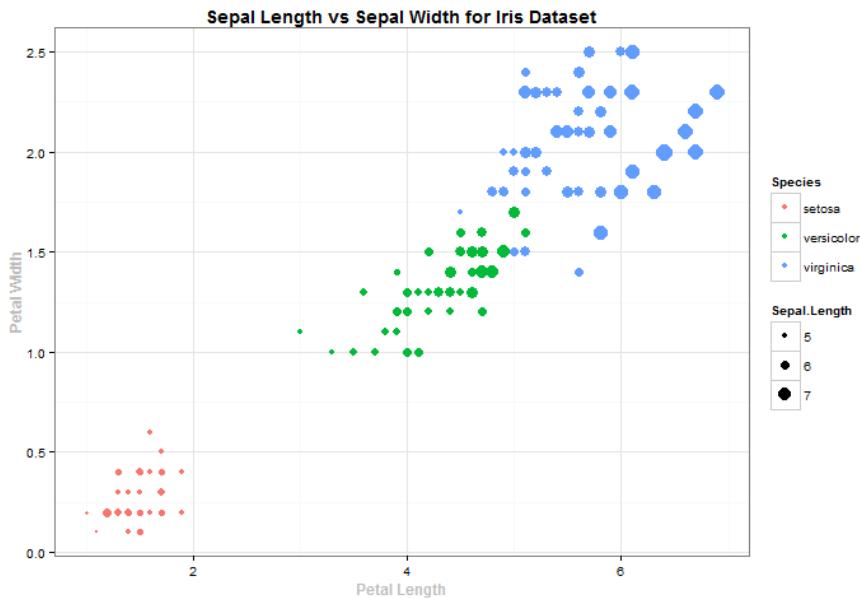


Figure 10.3: Bubble Plot

```

1 ggplot(data=iris, aes(x=Petal.Length, y=Petal.Width))+
  geom_point(aes(color=Species, size=Sepal.Length))+
  geom_smooth()+
  xlab("Petal Length")+
  ylab("Petal Width")+
  ggtitle("Sepal Length vs Sepal Width for Iris Dataset")

```

Listing 10.3: Bubble plot

Si agregamos el tamaño de los puntos para representar una variable numérica tenemos lo que se llama un *bubble plot*. En este caso agregamos el tamaño de los Sépalos para el tamaño de las burbujas. Esto probablemente no aporte mucha claridad al plot pero sirve para ver que Iris Virgínica parece tener sépalos, en general, mas grandes que las otras dos especies de flores.

Los plots de burbujas se hicieron famosos cuando Hans Rosling mostró su eficacia para transmitir información en una muy famosa charla de TED (The best stats you've ever seen). En esta charla Rosling usó plots de burbujas animados para mostrar la evolución de diferentes países de acuerdo a múltiples dimensiones al mismo tiempo. Por ejemplo usando en el eje *X* el nivel económico del país (GDP) y en el eje *Y* la tasa de mortalidad infantil, usando el tamaño de

las burbujas para representar la población de los países y el color para indicar su continente.

El golpe de gracia de la presentación de Rosling fue el uso de la animación para mostrar la evolución del plot de burbujas a lo largo del tiempo. Esto no solo agregó una dimensión extra sino también un importante impacto visual en donde podía, literalmente, verse como los países tendían hacia un mismo equilibrio.

En un Scatter Plot se puede agregar una línea de tendencia que ajuste a la relación entre las variables.

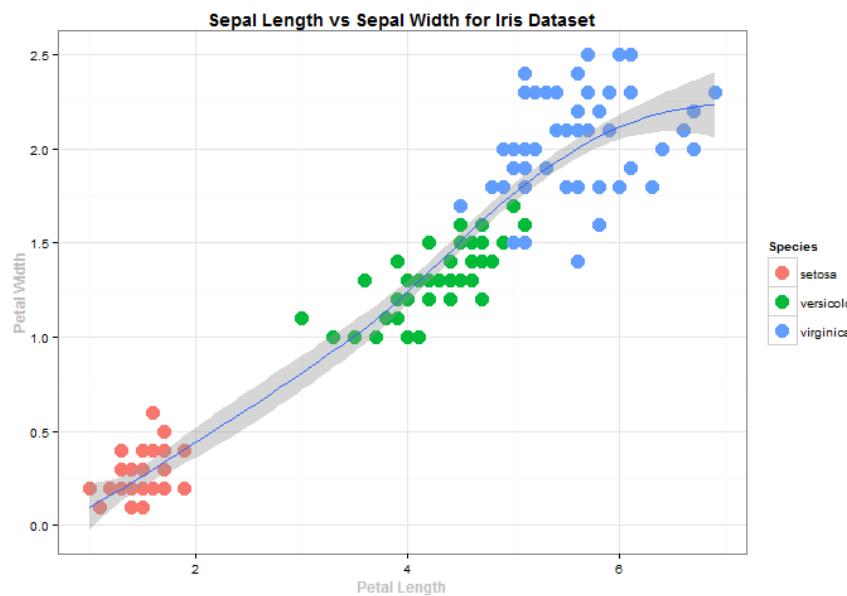


Figure 10.4: Scatter Plot con Tendencia

```

1 ggplot(data=iris ,aes(x=Petal.Length,y=Petal.Width))+
2   geom_point(size=5,aes(color=Species))+  

3   geom_smooth()  

4   xlab("Petal Length")+
5   ylab("Petal Width")+
6   ggtitle("Sepal Length vs Sepal Width for Iris Dataset")

```

Listing 10.4: Scatter Plot con Tendencia

La tendencia puede mostrarse de varias formas distintas, desde una simple regresión lineal hasta modelos polinómicos. Es común indicar mediante un área sombreada el grado de confianza que el modelo de tendencia tiene con respecto a los datos. En nuestro plot vemos que la confianza de predecir el ancho de los pétalos en función de su altura decrece para los pétalos más grandes y es muy

efectiva para la especie *iris versicolor* esto lo vemos porque la línea de tendencia es muy angosta al pasar por la zona con los puntos verdes.

Si quisieramos ver como la tendencia varía según la especie de flor podríamos *facetar* nuestro plot de acuerdo a la especie, esto equivale a hacer un scatter plot por cada especie y luego mostrarlos uno al lado del otro.

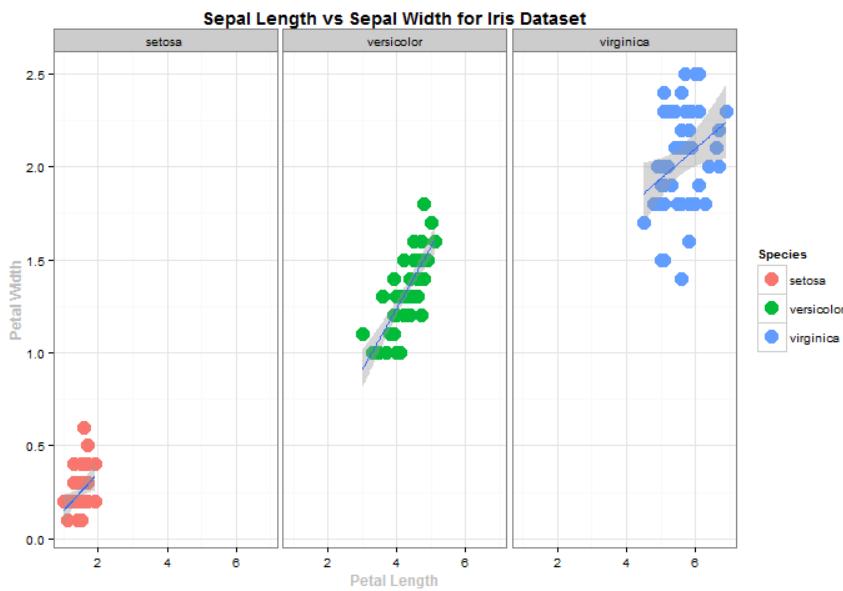


Figure 10.5: Scatter Plot con Tendencia

```

1 ggpplot(data=iris , aes(x=Petal.Length , y=Petal.Width))+
2   geom_point(size=5,aes(color=Species))+  

3   geom_smooth(method="lm")+
4   xlab("Petal Length")+
5   ylab("Petal Width")+
6   ggtitle("Sepal Length vs Sepal Width for Iris Dataset")+
7   facet_grid(.~Species)

```

Listing 10.5: Scatter Plot Facetado por Especie

Lo que hicimos fue "facetar" nuestro plot de puntos de acuerdo al valor de una variable categórica, en este caso al especie de la flor. Al hacer esto la línea de tendencia es ahora individual para cada plot y podemos observar que es muy efectiva para *iris versicolor* pero bastante pobre para *iris virginica*. Es decir que para *iris virginica* el alto de los pétalos no es tan buen predictor para el ancho de los mismos en comparación con las otras dos especies.

Es posible facetar lado a lado o con un plot debajo de otro. En general esto lo debemos decidir de acuerdo al tamaño de los ejes *X* e *Y*, cuando *X* es el eje

mas largo es conveniente facetar los plots uno debajo del otro y cuando ocurre lo contrario lado a lado, esto permite minimizar le grado de compresión visual de nuestro plot.

Finalmente veamos un ejemplo en el cual ploteamos el precio de diamantes en función de los carates, esta información está disponible en el set de datos "diamonds".

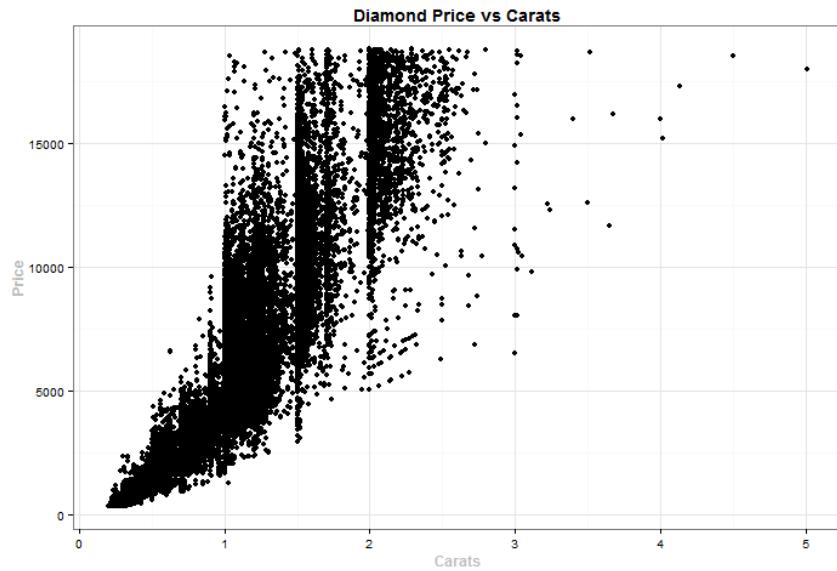


Figure 10.6: Scatter Plot para Diamonds

```

1 ggplot(data=diamonds, aes(x=carat, y=price))+
  geom_point(size=2)+
  xlab("Carats")+
  ylab("Price")+
  ggtitle("Diamond Price vs Carats")

```

Listing 10.6: Scatter Plot con Muchos Puntos

En este plot estamos mostrando un punto para cada par (carats,price) que existe en nuestro set de datos. En principio podemos ver que parece existir una tendencia lineal que indica que cuantos mas carates tiene el diamante mayor es su precio, esto es cierto hasta 1 carate en donde vemos que se produce un efecto muy curioso por el cual aparecen diamantes caros de forma casi independiente de la cantidad de carates de los mismos.

El problema del plot 8.3 es que como tenemos muchos puntos la superposición de los mismos no permite apreciar las zonas en las cuales hay mayor densidad de puntos. Para solucionar esto podemos usar un valor de transparencia (alpha) para cada punto:

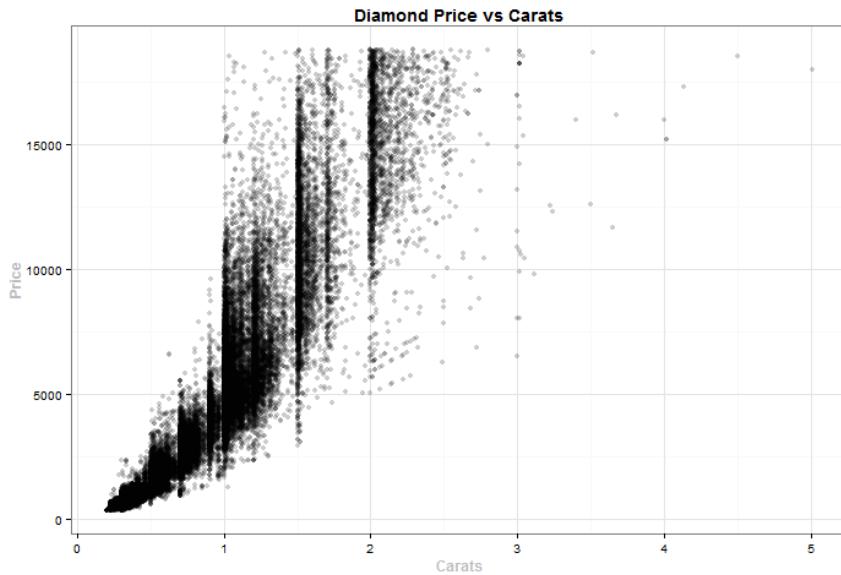


Figure 10.7: Scatter Plot Para Diamonds con Transparencia

```

1 ggplot(data=diamonds, aes(x=carat, y=price))+
2   geom_point(size=2, alpha=.2)+
3   xlab("Carats")+
4   ylab("Price")+
5   ggtitle("Diamond Price vs Carats")

```

Listing 10.7: Scatter Plot con Transparencia

Agregando la transparencia podemos ver como mejor nitidez las zonas en las que se concentran la mayoría de los diamants, vemos que hay muchos diamantes de 1, 1.5 o 2 carates y que los diamantes de mas de 2 carates son raros y de mas de 3 carates muy raros. Si cortamos verticalmente el gráfico por debajo del precio de 5000 dólares es muy difícil apreciar tendencia alguna, es decir que aprendemos que podemos usar carates para predecir el precio solo para diamantes de menos de un carate, luego de esto hay otros factores que seguramente son mas importantes.

Veamos que sucede si hacemos un scatter plot en donde el eje x es una variable categórica:

```

1 ggplot(diamonds, aes(x=cut, y=price, color=cut)) +
2   geom_point(alpha=0.3) +
3   xlab("Cut") +
4   ylab("Price") +
5   ggtitle("Diamonds Price by Cut")

```

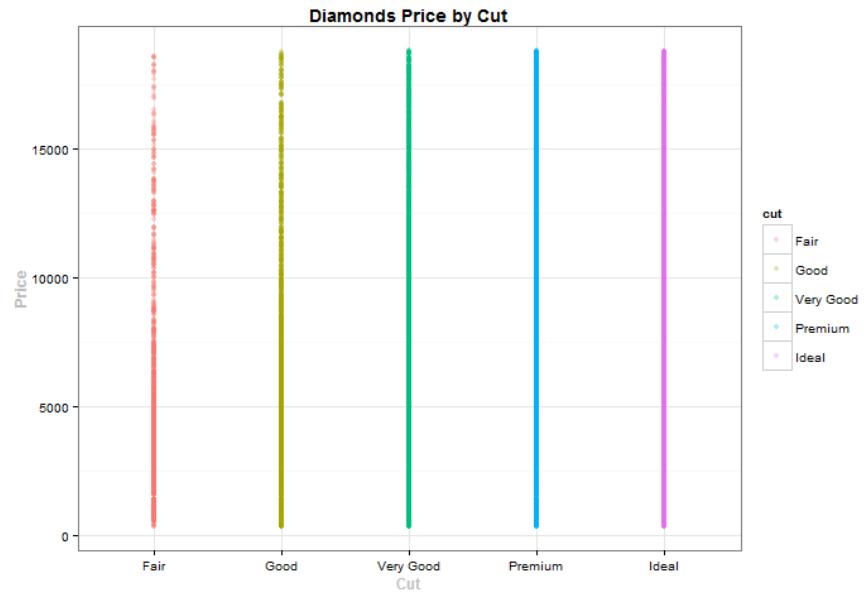


Figure 10.8: Scatter Plot Para Una Variable Categórica (MAL)

Listing 10.8: Scatter Plot para Variables Categóricas (MAL)

Este plot no funciona bien ya que en el eje x nuestros puntos solo pueden tomar 5 valores posibles por lo tanto solo tenemos unas líneas finitas que no tienen mucho sentido. Existe una forma de que este plot funcione que consiste en perturbar un poco cada punto, de esta forma no todos tendrán el mismo valor en x , a este proceso se lo denomina *jittering* y el resultado es el siguiente:

```

1 ggplot(diamonds, aes(x=cut, y=price, color=cut)) +
2   geom_jitter(alpha=0.3) +
3   xlab("Cut") +
4   ylab("Price") +
5   ggtitle("Diamonds Price by Cut")

```

Listing 10.9: Scatter Plot para Variables Categóricas con Jittering

En este plot vemos mucho mejor la forma en que se distribuyen los diamantes de acuerdo a su corte y precio. Es curioso que independientemente del corte existan gran cantidad de diamantes con un precio elevado aunque está claro que esto es recién visible a partir de "very good", vemos que la zona de máxima densidad en el precio aumenta al pasar de "fair" a "good" pero es mas difícil distinguir el efecto, si existiera al pasar de "very good" a "premium" o "ideal".

Resumimos ahora las características generales de un Scatter Plot que es un plot muy versátil y que permite mostrar información usando varias dimensiones



Figure 10.9: Scatter Plot Para Una Variable Categórica Usando Jittering

al mismo tiempo.

1. Los ejes x e y deben ser variables numéricas
2. Puede usarse el color para agregar una variable categórica o numérica
3. Puede usarse el tamaño de los puntos para agregar una variable categórica o numérica
4. Puede usarse transparencia para darle mayor claridad al plot
5. Puede agregarse una función de tendencia
6. Puede usarse la forma de los puntos para agregar una variable categórica.

10.4 Gráficos de Barras

Los gráficos de barras son muy populares, y constituyen una de las visualizaciones en donde mayor cantidad de errores se cometan. Hay dos principios básicos a observar cuando hacemos gráficos de barras: una de las variables tiene que ser categórica y los ejes deben comenzar en cero. La variable categórica es aquella para la cual vamos a crear las barras, una barra por cada valor posible de la variable.

Empecemos con un ejemplo muy simple usando el set de datos mtcars que tiene información sobre autos, creemos un gráfico de barras con la cantidad de autos que hay para las diferentes cantidades de cilindros posibles.

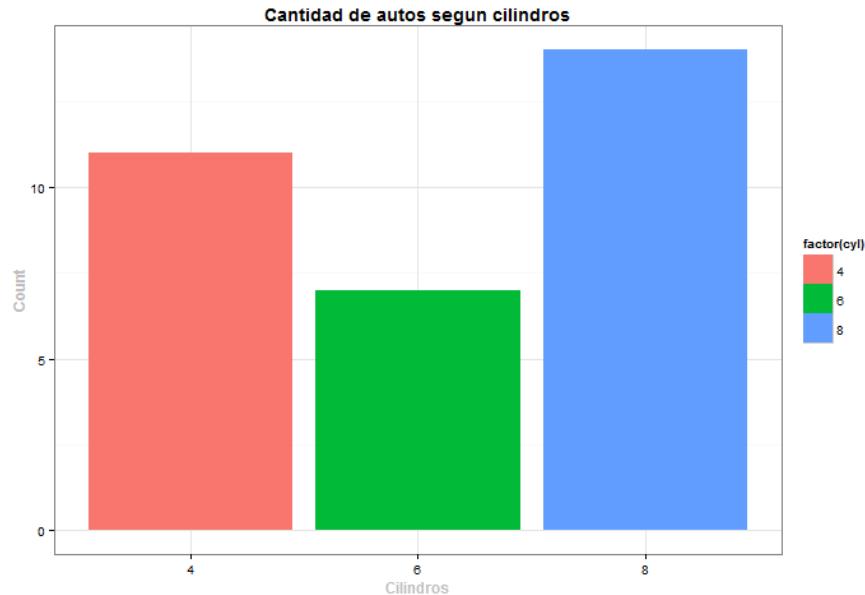


Figure 10.10: Plot de Barras para MtCars

```

1 ggplot(mtcars, aes(factor(cyl)))+
  geom_bar(aes(fill=factor(cyl)))+
  xlab("Cilindros")+
  ylab("Count")+
  ggtitle("Cantidad de autos segun cilindros")

```

Listing 10.10: Gráfico de Barras simple

Hemos usado el color simplemente para reforzar el concepto de que cada barra corresponde a una categoría diferente, no es algo necesario y es redundante con el eje x del plot pero mejora la visualización. Como podemos ver hay mas autos de 8 cilindros que 6 y 4 y los de 4 son mas populares que los de 6.

Por supuesto es posible dividir cada barra en zonas según una variable categórica, por ejemplo el tipo de corte de diamante para observar de que forma se compone la cantidad de cada uno de los diferentes niveles de claridad de los diamantes.

```

1 ggplot(diamonds, aes(clarity, fill=cut))+
  geom_bar()+
  xlab("Clarity")+
  ylab("Count")+

```

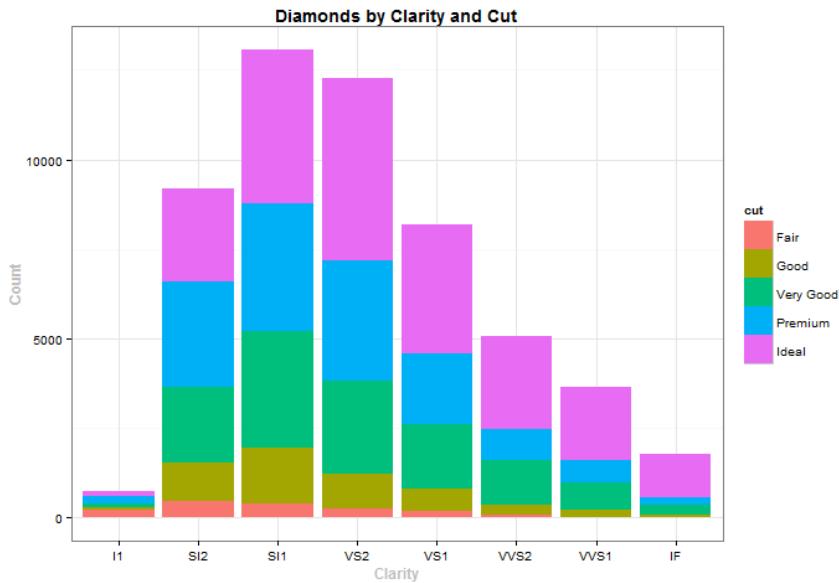


Figure 10.11: Plot de Barras Apiladas

```
5 ggtile("Diamonds by Clarity and Cut")
```

Listing 10.11: Plot de Barras Apiladas

En un gráfico con barras apiladas la altura de cada barra es el total para la categoría indicada en el eje X y cada barra se sub-divide en alturas proporcionales de acuerdo a otra variable categórica que en nuestro caso es el corte del diamante.

Podemos ver por ejemplo que para todos los niveles de claridad existe una buena cantidad de diamantes con corte "premium" y en general parece ser que el tipo de corte no depende de la claridad excepto para "fair" y "good" en donde notamos que a medida que aumenta la claridad la cantidad de diamantes correspondientes a esos cortes disminuye.

Una variante es forzar a que todas las barras ocupen la misma altura, esto lo podemos hacer creando una variable adicional que indique que porcentaje del total de cada valor de la variable categórica toma la otra variable.

```
1 ggplot(diamonds, aes(clarity, fill=cut)) +
  geom_bar(position="fill") +
  xlab("Clarity") +
  ylab("Count") +
  5 ggtile("Diamonds by Clarity and Cut")
```

Listing 10.12: Plot de Barras Apiladas completas

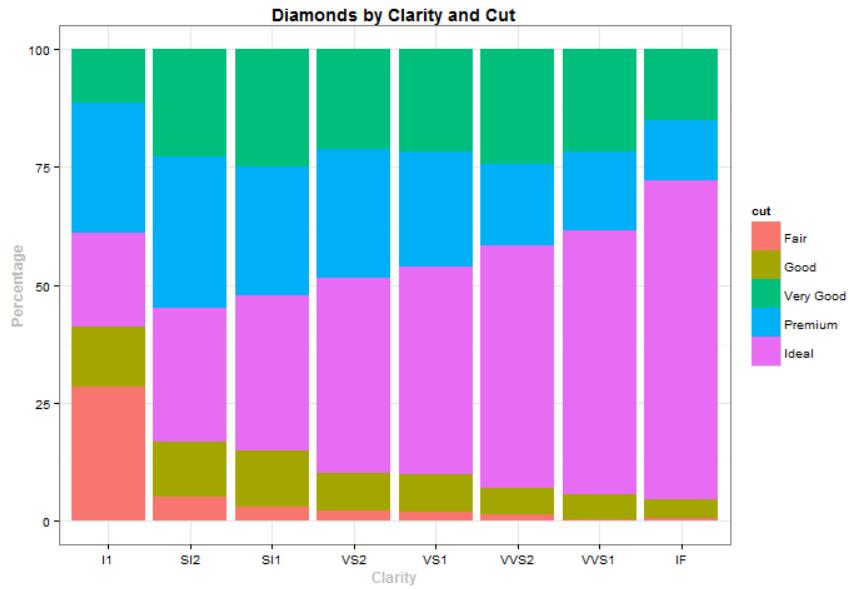


Figure 10.12: Plot de Barras Apiladas Completas

Podemos ver que ahora el eje *Y* va de 0 a 100 indicando un porcentaje, en este plot ya no tenemos información sobre el total de diamantes por cada nivel de claridad pero es mas simple comparar la cantidad de diamantes de cada corte según el nivel de claridad. Es decir que perdemos información pero ganamos claridad en otro aspecto. Aquí podemos ver muy claramente que a medida que aumenta la claridad la cantidad de diamantes con corte "fair" o "good" disminuye. Esto no ocurre con los cortes que son "very good" o superior.

Otra variante es mostrar las barras una al lado de la otra en lugar de apiladas:

```

1 ggpplot(diamonds, aes(clarity, fill=cut)) +
2   geom_bar(position="dodge") +
3   xlab("Clarity") +
4   ylab("Count") +
5   ggtitle("Diamonds by Clarity and Cut")

```

Listing 10.13: Plot de Barras Contiguas

En esta variante el plot podemos ver la cantidad de diamantes para cada combinación de claridad y corte. Por cada nivel de claridad tenemos tantas barras como cortes existen. Aquí vemos claramente que para la claridad V82 en adelante el corte "ideal" empieza a ser predominante, esto quiere decir que para diamantes con gran claridad no se hacen trabajos de corte económicos sino que se busca la perfección. Para diamantes con niveles de claridad pobres la distribución de los cortes es mas pareja, es decir que puede haber tantos cortes

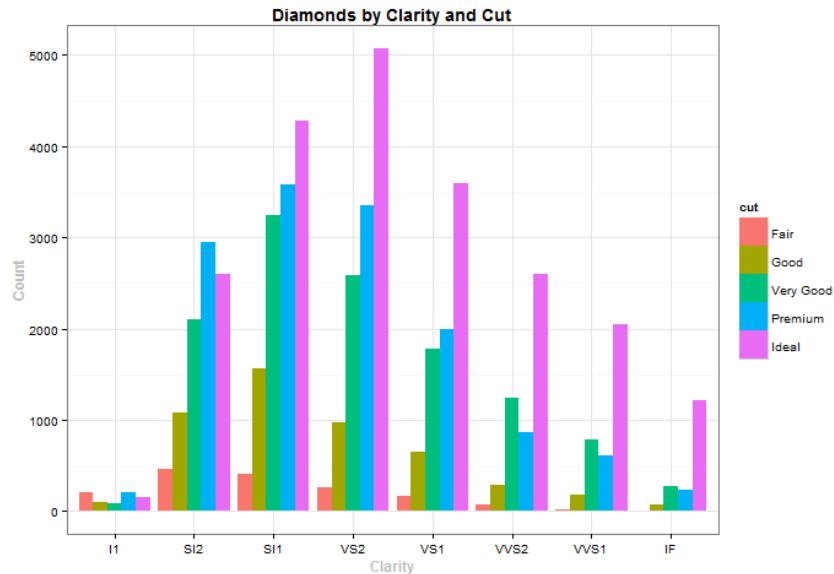


Figure 10.13: Plot de Barras Contiguas

”fair” como ”ideal” en cantidades similares para un diamante con el nivel de claridad mas bajo posible.

Hay que tener cuidado con la cantidad de barras dentro de cada ”columna” de nuestro plot, si son muchas la interpretación del mismo puede volverse dificultosa. Observemos como este tipo de plot es en realidad un plot de barras en donde cada barra es otro plot de barras.

Es común en ciertos medios manipular la percepción de la audiencia cambiando la escala de los ejes de un gráfico de barras. Cuando los ejes no comienzan en cero un plot de barras puede mostrar una diferencia sustancial en el valor de dos variables cuando en realidad en la versión completa (y correcta) del plot se verían muy parecidas.

10.5 Histogramas y Plots de Densidad

Un histograma sirve para mostrar la distribución de una determinada variable, es decir la cantidad de veces que la variable toma determinados valores. Para construir un histograma hacen falta dos parámetros: la variable en cuestión que tiene que ser numérica (continua o discreta) y el ancho que van a tener las columnas del histograma. Por ejemplo si una variable puede tomar valores continuos entre 0 y 10 y queremos que nuestro histograma tenga 20 columnas entonces el ancho de las mismas será de 0.5. Nuestro histograma tendrá entonces

la cantidad de veces que la variable toma valor entre 0 y 0.5, luego entre 0.5 y 1.0, etc.

Es importante destacar que un histograma, si bien muestra barras, es completamente diferente a un plot de barras. En el histograma la variable x es numérica mientras que en el plot de barras es categórica, en un histograma el eje x siempre está ordenado mientras que en un plot de barras puede tener cualquier orden. En un histograma no puede haber espacios entre las barras y en un plot de barras sí, el eje y en un plot de barras puede ser cualquier valor numérico mientras que en un histograma siempre es una cantidad.

Nuestro primer histograma muestra la distribución de la variable carats en el set de datos diamonds.

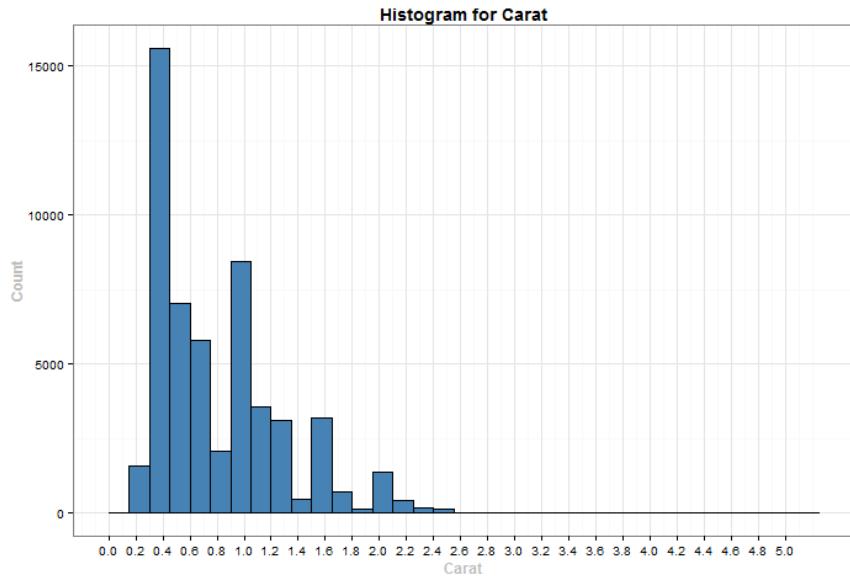


Figure 10.14: Histograma para Carats

```

1 ggplot(diamonds, aes(x=carat)) +
  geom_histogram(color="black", fill="steelblue", binwidth=.15) +
  xlab("Carat") +
  ylab("Count") +
  scale_x_continuous(breaks=seq(0,5,.2)) +
  ggtitle("Histogram for Carat")

```

Listing 10.14: Histograma para Carats

El histograma nos muestra la distribución de la variable "carats" el eje X muestra los posibles valores de la variable discretizados de acuerdo a ciertos intervalos y el eje Y indica la cantidad de veces que la variable cayó dentro de cada uno de esos intervalos para nuestro set de datos. En el caso de carats vemos

que los valores menores a 1.5 son los mas populares pero hay algunos diamantes que pueden llegar hasta 5, esta es una distribución *right skewed* porque presenta una cola larga hacia la derecha.

El color del histograma es simplemente decorativo y no debería nunca tener colores diferentes para cada barra. Si cambiamos el parámetro *binwidth* podemos crear un histograma con mas o menos granularidad.

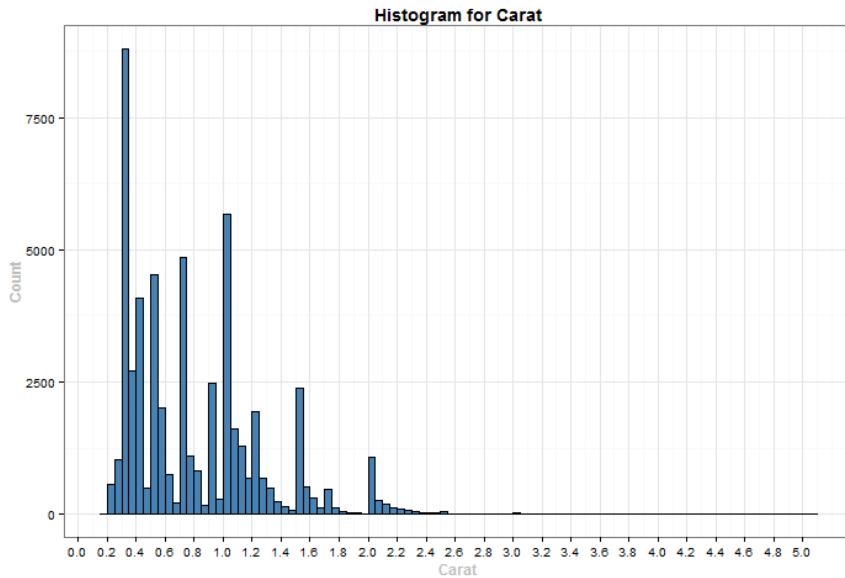


Figure 10.15: Histograma para Carats con mas bins

```

1 ggplot(diamonds, aes(x=carat)) +
2   geom_histogram(color="black", fill="steelblue", binwidth=.05) +
3   xlab("Carat") +
4   ylab("Count") +
5   scale_x_continuous(breaks=seq(0,5,.2))+
6   ggtitle("Histogram for Carat")

```

Listing 10.15: Histograma para Carats con mas bins

En la segunda versión discretizamos carats en una mayor cantidad de intervalos (bins) en donde antes teníamos una sola barra ahora tenemos varias mostrando valores mas precisos de la variable, en general no es necesario tener una gran cantidad de bins ya que con unos 10 a 20 bins suele alcanzar para entender la distribución de la variable. El tamaño de cada bin puede calcularse como $(max - min)/k$ siendo k la cantidad de bins que queremos mostrar en el histograma.

Un plot de densidad es una versión continua de un histograma, aquí no hace falta indicar el tamaño de los bins. Lo que se muestra es como se distribuye la

densidad de la variable numérica a lo largo de todos sus valores posibles.

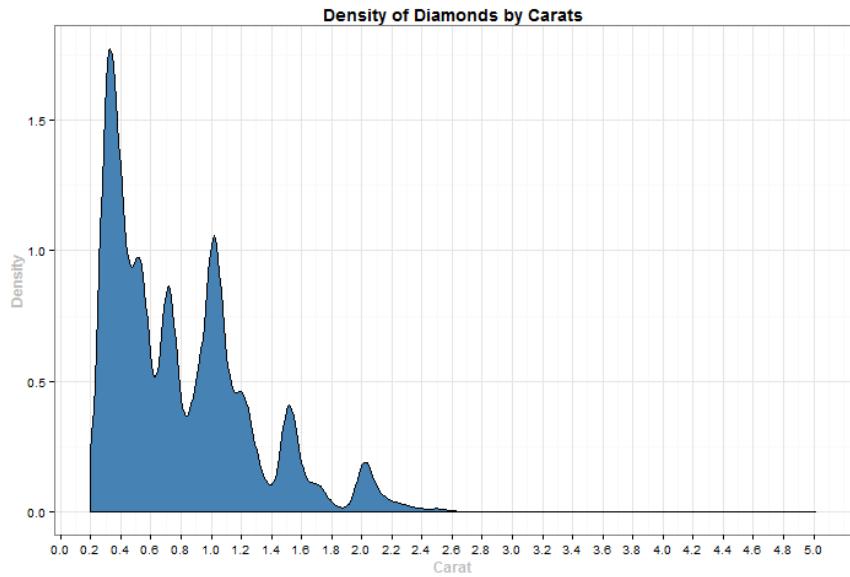


Figure 10.16: Plot de Densidad para Carats

```

1 ggplot(diamonds, aes(x=carat)) +
2   geom_density(fill="steelblue") +
3   xlab("Carat") +
4   ylab("Density") +
5   scale_x_continuous(breaks=seq(0,5,.2)) +
6   ggtitle("Density of Diamonds by Carats")

```

Listing 10.16: Plot de Densidad para Carats

Aquí es mas evidente el efecto del *long tail* hacia la derecha, debemos destacar que si el eje *X* llega hasta el valor 5.0 es porque en algún momento la variable carats toma este valor, sin dudas un diamante muy valioso.

Este histograma nos muestra picos entorno a los valores de carats 0.5, 1.0 y 1.5 probablemente porque se trata de valores populares por el redondeo. Cuando tenemos estos múltiples picos en un plot de densidad siempre tenemos que considerar cuál es el motivo por el cual los datos presentan picos para ciertos valores en lugar de distribuirse de forma continua. En general la explicación es que la variable no es realmente continua sino que es semi-continua, es decir que tiene una cierta cantidad de valores que toma con mas frecuencia que otros, esto es equivalente, en cierta forma, a una discretización natural de la variable.

Podemos superponer varios plots de densidades o histogramas usando una variable categórica como separador.

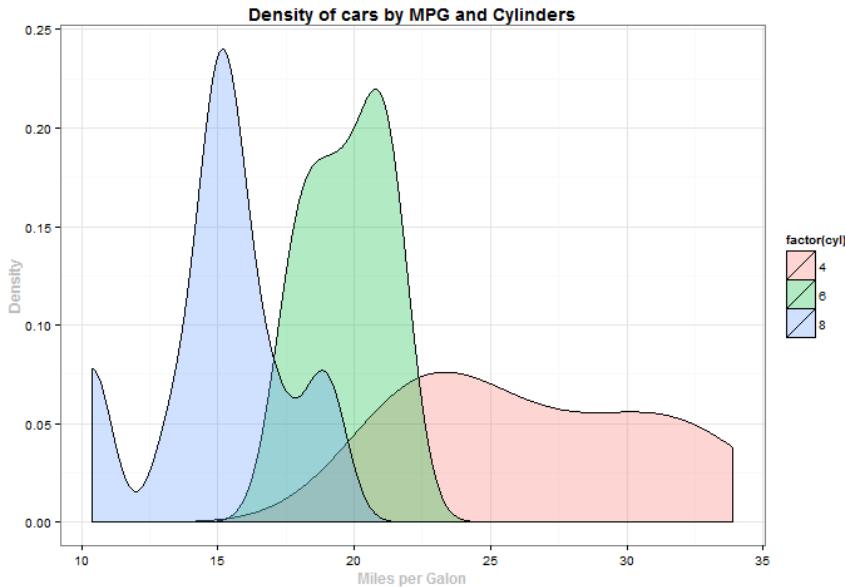


Figure 10.17: Plot de Densidad para MPG por cilindros en Mtcars

```

2 | ggplot (mtcars , aes (x=mpg, fill=factor ( cyl ))) +
3 |   geom _density (alpha=0.3) +
4 |   xlab (" Miles per Galon" ) +
5 |   ylab (" Density" ) +
6 |   ggttitle (" Density of cars by MPG and Cylinders" )

```

Listing 10.17: Plot de Densidad para MPG por cilindros en MtCars

Aquí vemos que cantidad de autos tenemos en mtcars según los distintos valores que puede tomar la variable "mpg" (consumo) de acuerdo a la cantidad de cilindros del auto. Observamos que a mayor cantidad de cilindros mayor es el consumo.

Vemos también la distribución prácticamente normal de los autos de 6 cilindros y una distribución semi-uniforme para los de 4 cilindros, los autos de 8 cilindros presentan una extraña distribución que parece ser bimodal o trimodal (tiene dos o tres picos según como la interpretemos).

10.6 Boxplots

Un *boxplot* es una forma de visualizar la distribución de una variable numérica, en general se usa para comparar la distribución de la variable de acuerdo a una variable categórica mostrando varios boxplots uno al lado del otro en el eje x.

Veamos un ejemplo rápido de un boxplot para MtCars.

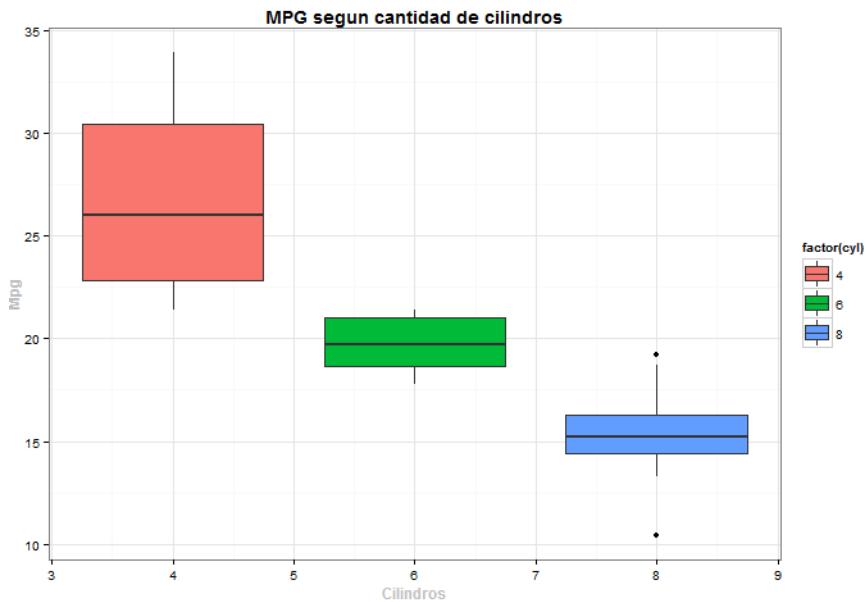


Figure 10.18: Boxplot para mpg según cantidad de cilindros

```

1 ggpplot(mtcars, aes(x=cyl, y=mpg, fill=factor(cyl))) +
2   geom_boxplot() +
3   xlab("Cilindros") +
4   ylab("Mpg") +
5   ggtitle("MPG según cantidad de cilindros")

```

Listing 10.18: Boxplot para mpg según cantidad de cilindros

El color en este caso no es necesario ya que el eje X indica la cantidad de cilindros, esto sigue los principios básicos que listamos al comenzar el capítulo ya que en blanco y negro la interpretación del plot sería la misma. El eje Y nos da una idea de los valores que puede tomar la variable mpg en nuestro set de datos.

El plot nos muestra que el rendimiento (mpg) es mayor para los autos de 4 cilindros, también podemos ver que los autos de 4 cilindros tienen una distribución mas "amplia" ya que el tamaño de su boxplot es mayor. Expliquemos cuáles son los elementos en un boxplot para volver a analizar nuestro plot.

Listing 10.19: Elementos de un boxplot

La "caja" del boxplot va desde el primer al tercer cuantil, es decir que el 25% de los datos están por debajo de la caja y el 25% de los datos están por encima de la caja. La caja concentra entonces el 50% de los datos y la línea

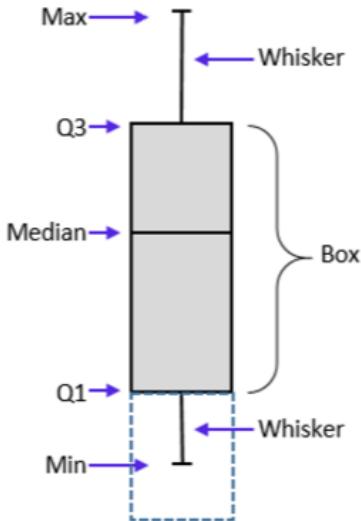


Figure 10.19: Elementos de un Boxplot

dentro de la caja muestra la media. Las líneas que salen de la caja van desde el primer cuartil hasta el valor mínimo y máximo, los puntos son valores anómalos (outliers).

Regresando al boxplot que hicimos vemos que para 4 cilindros vemos que hay mas autos desde la mediana al tercer cuantil que desde la mediana al primer cuantil es decir que hay menor variabilidad en el consumo por debajo de la mediana que por encima. Esto indica una variable con un leve left skew. Para los autos de 8 cilindros vemos que hay valores anómalos, hay un auto que tiene un consumo excesivo y otro que tiene un consumo muy bueno para ser un auto de 8 cilindros. Vemos también que para los autos de 6 cilindros la distribución es muy pareja, prácticamente normal.

Listing 10.20: Boxplot y plot de densidad

La comparación entre el boxplot y el plot de densidad nos muestra que ambos plots contienen prácticamente la misma información, observemos el "pico" de la distribución para 8 cilindros que en el boxplot aparece como un outlier.

Para 6 cilindros la distribución es prácticamente normal y eso también se ve en el boxplot, también vemos que tiene poca variabilidad porque hay muy poca distancia entre el mínimo y el máximo. Para 4 cilindros el boxplot nos dice que la distribución es mas "ancha" y que está volcada hacia la izquierda cosa que vemos en el plot de densidad.

Un boxplot puede mostrar si una variable presenta skewing pero no puede

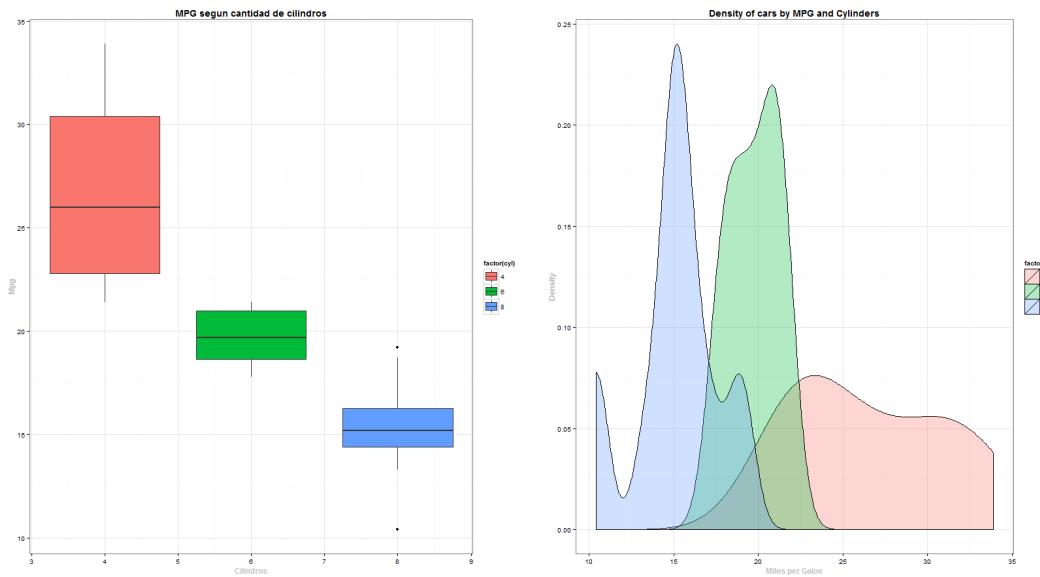


Figure 10.20: Boxplot y Plot de Densidad

indicar la modalidad de la misma. En el boxplot para 8 cilindros no podemos darnos cuenta que se trata de una variable trimodal.

La ventaja de un boxplot es que podemos plotear una gran cantidad de boxplots uno al lado del otro para comparar de acuerdo a una variable categórica con muchos valores caso en el cual un plot de densidad, incluso usando transparencia, sería bastante confuso. Este es un recurso muy útil para entender si una variable afecta la distribución de otra.

Veamos para "clarity" en diamonds como se distribuye el precio.

```

2 ggplot(diamonds , aes(x=clarity ,y=price , fill=factor( clarity ))) +
3   geom_boxplot()+
4     xlab("Clarity") +
5     ylab("Price") +
6     ggttitle(" Price by Clarity for Diamonds")

```

Listing 10.21: Boxplot para Clarity vs Price

El boxplot nos da al mismo tiempo información de como se distribuye el precio de los diamantes para todos los valores posibles de "clarity". Como podemos ver hay muchos outliers en todos los casos y siempre hacia arriba, es decir que independientemente de su "claridad" es posible que un diamante tome un valor muy alto. Los precios inusualmente bajos son en cambio prácticamente nulos, es decir que no hay diamantes muy baratos cosa que podríamos llegar a sospechar.

Vemos que la distribución en algunos lados es muy despareja, la presencia

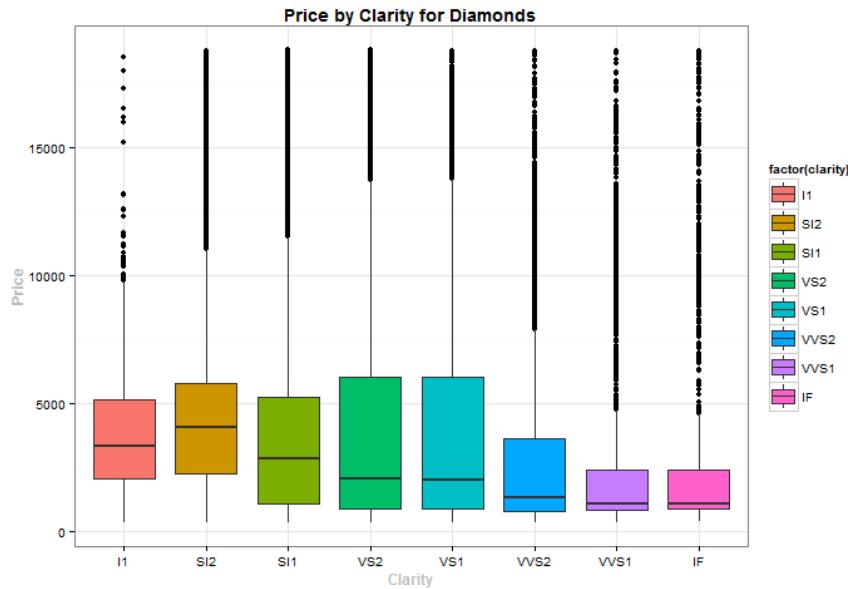


Figure 10.21: Boxplot para Clarity vs Price

de los outliers hace que en todos los casos tengamos un fuerte "right skew" la mayoría de los valores son precios bajos pero hay un "long tail" de precios muy altos, es decir diamantes muy valiosos.

La mediana nos indica que el valor medio de un diamante no depende de su claridad, el valor medio mas alto está dado para un nivel de claridad bajo (SI2) y se debe probablemente a la gran cantidad de outliers que presenta esta categoría. Los niveles de claridad mas altos muestran cajas de menor altura, mas compactas lo cual denota distribuciones con menor variabilidad.

Para clarificar la situación es muchas veces una buena idea acompañar un boxplot de un plot de densidad o incluso de un scatter plot usando jittering. Estos plots al ser standard se pueden generar rápidamente para el análisis exploratorio de datos.

```

1 ggplot(data=diamonds, aes(x=clarity, y=price, color=clarity))+
  geom_jitter(size=2, alpha=.2)+
  xlab("Clarity")+
  ylab("Price")+
  ggtitle("Diamond Price vs Clarity")

```

Listing 10.22: Scatter plot para Price vs Clarity

En el scatter plot (usando jittering) vemos la confirmación de lo que se deduce en el boxplot, para todas las claridades la mayoría de los diamantes tiene un precio bajo pero con varios valores altos.

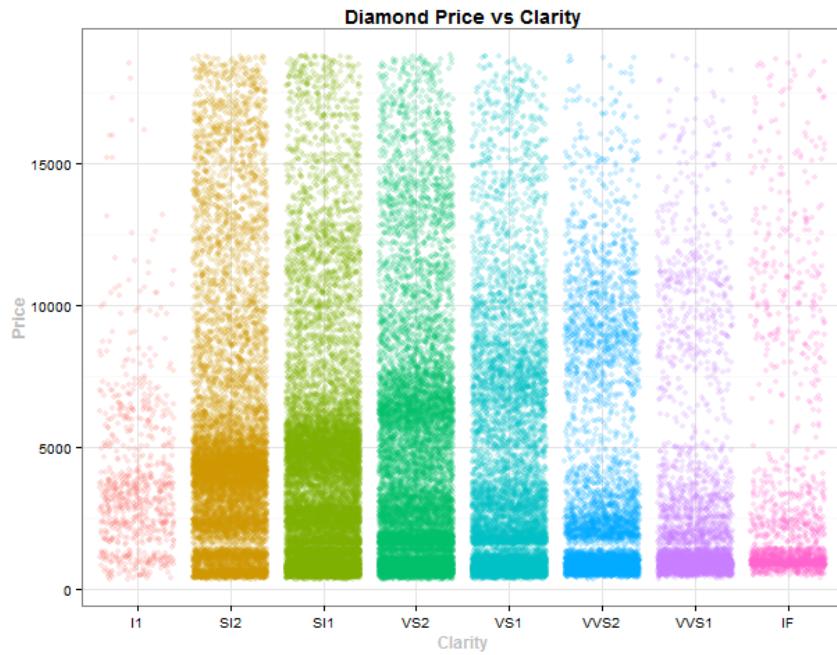


Figure 10.22: Violin Plot

Vemos también que el efecto es menor en "I1" que en el boxplot vemos que tiene menos outliers. El caso de la claridad "IF" en donde el boxplot nos mostró menor variabilidad queda ahora mas claro, hay menos cantidad de diamantes "IF" y la mayoría de ellos tiene un precio bajo. Es curioso e interesante ver que para todas las claridades excepto I1 hay una zona de precios bajos con una cantidad importante de diamantes. Como curiosidad vemos también una cierta separación en el rango de precios entre los diamantes que están en el rango mas bajo y todos los demás, esto podría deberse a la existencia de algún tipo de convención de precios para los diamantes mas económicos.

Todas estas hipótesis que surgen de estos plots son parte del resultado del análisis exploratorio de los datos, luego en caso de que nos resulte conveniente podemos aplicar algoritmos para verificar estas hipótesis.

En algunos casos cuando los outliers son muchos el boxplot puede no permitir comparar claramente cuál de las variables tiene mas o menos outliers, existe una variante del boxplot que soluciona esto denominada *plot de violín*

```

1 ggplot(diamonds, aes(x=clarity ,y=price , fill=factor(clarity))) +
  geom_violin()+
  xlab("Clarity") +
  ylab("Price") +
  ggtitle("Price by Clarity for Diamonds") +
  my_theme

```

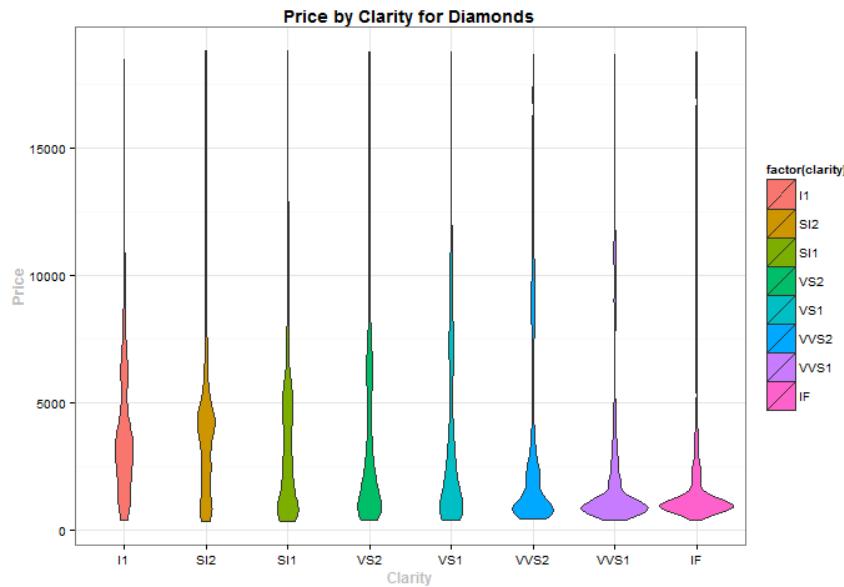


Figure 10.23: Violin Plot

Listing 10.23: Violin plot para Price vs Clarity

El plot de violín nos muestra un ancho correspondiente a la densidad de la variable para el valor del eje Y correspondiente. Es mucho mas sencillo interpretar el histograma de la variable en base al plot de violín en todos los casos tendremos "right skewing" pero en el caso de las claridades a la derecha la desproporción es mucho mayor. Vemos también que para "I1" la cantidad de valores anómalos es menor que para los otros valores de clarity.

En el plot de violín el ancho es un elemento visual importante a diferencia de un boxplot tradicional. El color tampoco es importante, el mismo plot de violín en blanco y negro sería efectivo. El rango de los ejes es igual al boxplot.

En general puede ser buena idea realizar tanto el boxplot como el plot de violín, estos plots junto un plot de densidad por cada categoría nos ayudarán a entender perfectamente las características estadísticas de una variable.

Es posible agregar una variable categórica con pocos valores posibles extra en un boxplot cuando los resultados no se tornan confusos, en este caso usando el color para distinguir entre dos variables posibles: caja automática o manual.

```

2 ggplot(mtcars, aes(factor(cyl), mpg)) +
  geom_boxplot(aes(fill=factor(am))) +
  xlab("Cilindros") +
  ylab("Mpg") +

```

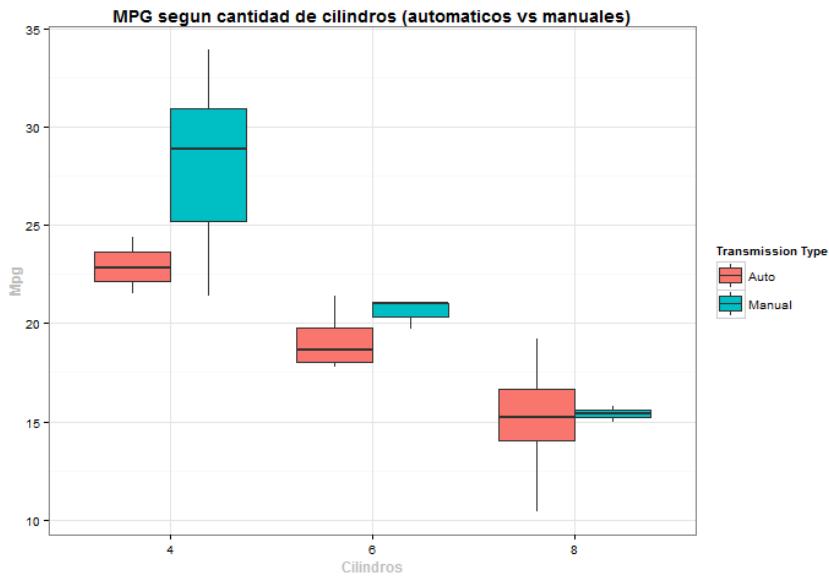


Figure 10.24: Boxplot según una segunda variable categórica

```
ggtile("MPG segun cantidad de cilindros (automaticos vs manuales)")
```

Listing 10.24: Boxplot con una segunda variable categórica

En el plot vemos en rojo los autos automáticos y en azul los manuales, vemos que para 4 cilindros los autos manuales tienen mejor rendimiento (menor consumo), esta tendencia se verifica también para 6 cilindros. Para 8 cilindros la tendencia desaparece y es aparentemente igual tener un auto manual o automático si nos preocupa el consumo, esto tal vez se deba a que un motor de gran cilindrada consume tanto que el efecto del tipo de transmisión es despreciable.

10.7 Gráficos de Líneas

Los gráficos de líneas son muy útiles, pero tienen una muy fuerte limitación, en casi todos los casos el eje X debe ser tiempo. Por lo tanto es un plot dedicado casi exclusivamente a series temporales. La variable en el eje Y puede ser de cualquier tipo aunque casi siempre es ordinal, ya sea numérica (el caso más frecuente) o categórica. El eje X representa entonces la evolución del valor de dicha variable a lo largo del tiempo.

Veamos un ejemplo para el precio de la acción IBM a lo largo del tiempo:

```
1 ggplot(ibm, aes(Date, Close)) +  
  geom_line(size=1.1, color="steelblue") +
```

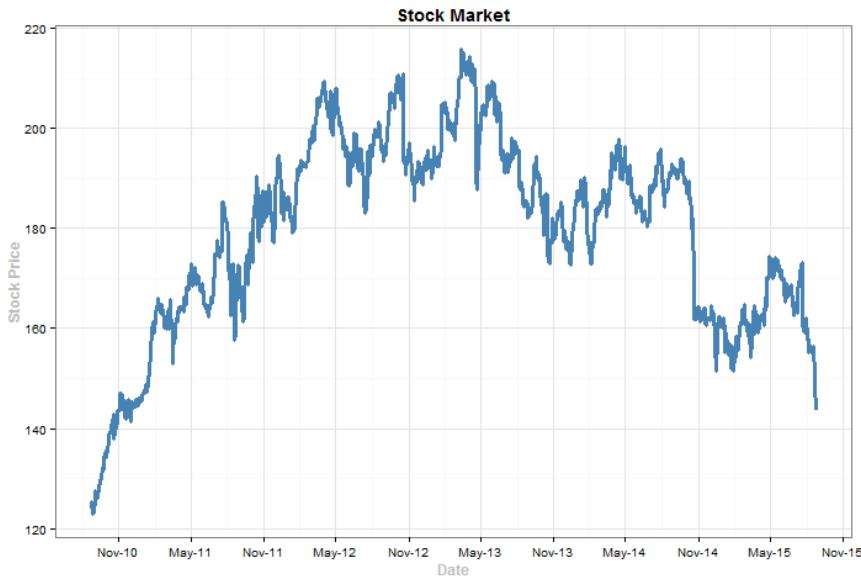


Figure 10.25: Plot de líneas para las acciones de IBM

```

3   xlab("Date")+
4   ylab("Stock Price")+
5   ggtitle("Stock Market")+
6   scale_x_date(labels = date_format("%b-%y"),breaks = date_
breaks("6 months"))

```

Listing 10.25: Plot de líneas para las acciones de IBM

Vemos que la tendencia de IBM fue alcista hasta aproximadamente mayo de 2013 y luego inició un descenso, dentro de esta tendencia la variable tiene zonas donde crece y zonas donde decrece, esto es muy común en el mercado de valores y en general en muchas variables económicas donde el crecimiento de una variable se explica mediante el efecto acumulativo de varias alzas y bajas.

Cuando nuestro set de datos tiene el formato (date,name,value) y name es una variable categórica podemos usar la misma para plotear varias líneas al mismo tiempo usando diferentes colores.

```

2   ggplot(stock ,aes(Date ,Close ,color=stock )) +
3     geom_line(size=1.1) +
4     xlab("Date")+
5     ylab("Stock Price")+
6     ggtitle("Stock Market")+
7     scale_x_date(labels = date_format("%b-%y"),breaks = date_
breaks("6 months"))

```

Listing 10.26: Plot de líneas para varias acciones



Figure 10.26: Plot de líneas para varias acciones

El plot compara las acciones de IBM, Apple y Linkedin durante 5 años, notemos que el plot de IBM tiene la misma forma pero ha cambiado en función de la escala porque APPLE tenía un valor para su acción mucho mas alto hasta que se desplomo en Junio de 2014. Podemos ver que la tendencia de Apple muestra en principio una cierta recuperación que vuelve a caer, IBM es la acción con valor mas estable y Linkedin tiene un comportamiento bastante variable.

Una opción interesante es usar el eje Y para representar el ranking de una variable en un cierto instante de tiempo. Por ejemplo:

La figura 10.27 nos muestra un gráfico de líneas para la popularidad de lenguajes en StackOverflow, en donde el eje Y representa el ranking del lenguaje a lo largo del tiempo. Este tipo de gráficos son mucho mas legibles que la alternativa de representar la cantidad de preguntas o tags de cada lenguaje a lo largo del tiempo.

Al usar el ranking en el eje Y se pierde la escala entre los valores de las diferentes categorías. Por ejemplo si la diferencia entre el primero y el segundo es mucho mayor que la diferencia entre el segundo y el tercero en este tipo de plots no se podría apreciar dicha diferencia.

Este tipo de plots pueden quedar muy bien en versiones interactivas en las que el usuario pueda seleccionar algunas de las variables a mostrar y que estas aparezcan en colores en el plot dejando a las otras en un tono de background, similar a lo que vemos en la figura 10.27

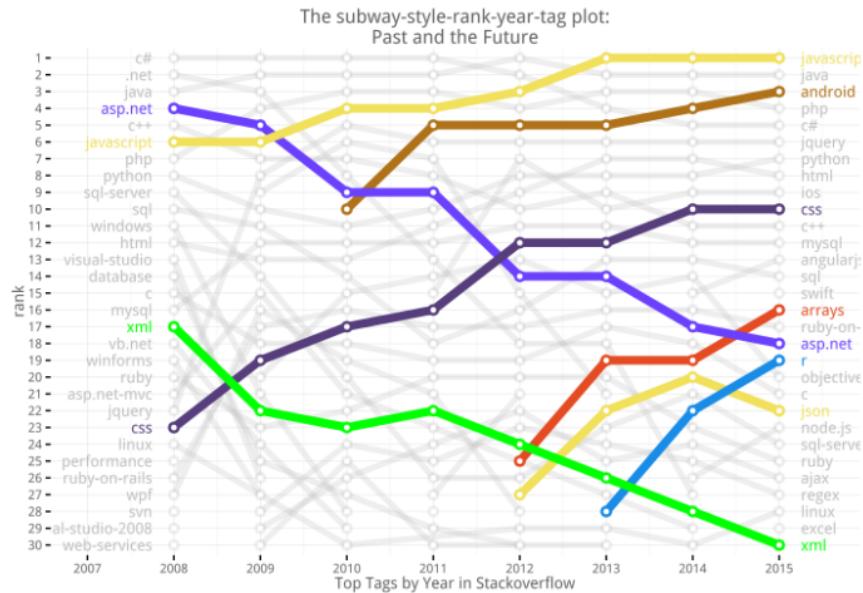


Figure 10.27: Gráficos de Líneas usando el eje Y para mostrar Ranking

En todos los casos el gráfico se lee de izquierda a derecha y los valores de interés deben leerse en el eje Y , cuando alguna de las variables está muy fuera de escala en relación a las demás se produce un efecto de compresión visual que puede dificultar la visualización del plot. En nuestro caso APPLE domina los valores del eje T lo cual hace que el plot de IBM parezca mucho mas plano de lo que es realmente como vimos en el plot anterior.

Los plots de líneas son extremadamente sensibles a valores erróneos o fuera de escala, basta que en algún momento alguna de las variables en el plot tome un valor inusualmente alto para que todo el plot quede comprimido a algo que no tiene sentido visual alguno.

10.8 Gráficos de área

Una variante de los gráficos de línea son los gráficos de área en donde el eje X sigue siendo el tiempo y el eje Y sigue siendo una variable pero ahora mostramos el área debajo de cada curva de forma proporcional al valor de la misma. Esto NO es lo mismo que colorear el área debajo de las curvas ya que el área debe ser proporcional al valor de la variable según alguna variable categórica.

Veamos como ejemplo un plot de área para la cantidad de películas por categoría a lo largo del tiempo.

```
ggplot(mtypes, aes(x=year, y=total, fill=category)) +
```

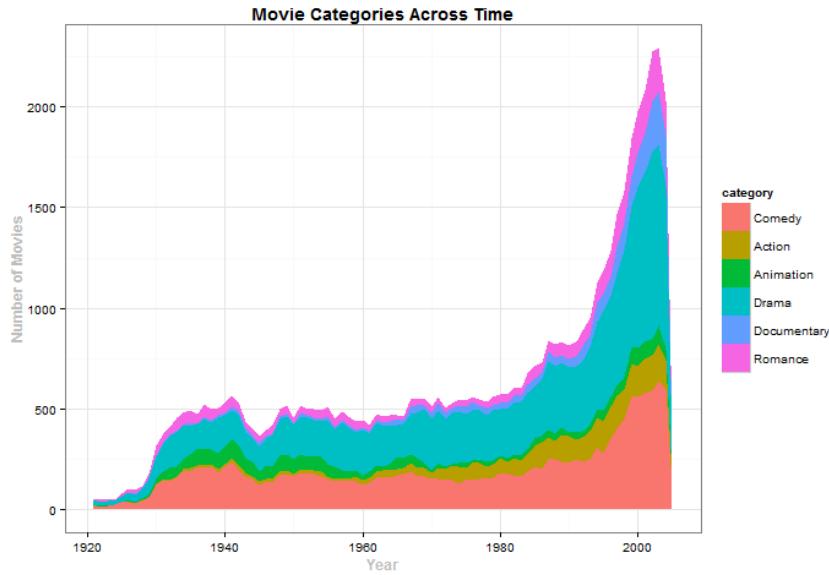


Figure 10.28: Plot de áreas para películas

```

2 |   geom_area()+
3 |   xlab("Year")+
4 |   ylab("Number of Movies")+
5 |   ggtitle("Movie Categories Across Time")

```

Listing 10.27: Plot de áreas para películas

En el plot podemos ver que las categorías mas populares son "drama" y "comedia", las películas de acción se vuelven mas populares luego de los 80. Podemos ver también el auge de las películas de animación durante el período entre 1940 y 1960 y su resurgimiento en 2000. En este plot también parece notarse un aumento en la cantidad de documentales en los últimos años.

Observemos la diferencia con un plot de líneas

```

1 | ggpplot(mtypes, aes(x=year ,y=total ,color=category))+ 
2 |   geom_line(size=1.1)+ 
3 |   xlab("Year")+
4 |   ylab("Number of Movies")+
5 |   ggtitle("Movie Categories Across Time")

```

Listing 10.28: Plot de líneas para películas

Notemos que las películas románticas están en la parte superior del plot de áreas pero su área es realmente pequeña, las películas mas populares son las dramáticas como podemos ver en el plot de líneas y por eso son las que mayor área tienen en el plot de áreas.

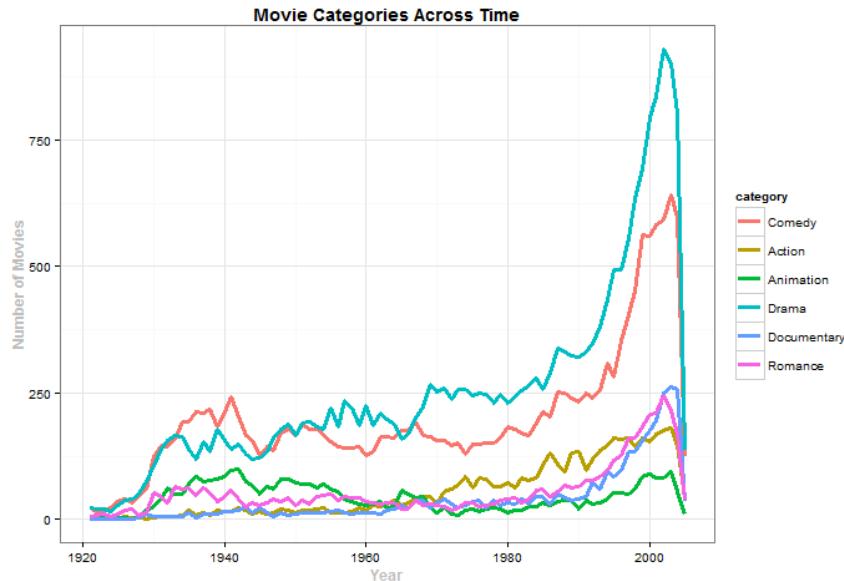


Figure 10.29: Plot de líneas para películas

Es decir que en un plot de áreas el valor superior de la última curva indica el acumulado para TODAS las categorías es decir cantidad total de películas y el área de cada categoría indica de forma proporcional su cantidad. Notemos que en el plot de áreas el eje Y toma valores mayores que en el plot de líneas. En el plot de líneas el valor máximo en el eje Y corresponde al máximo para alguna categoría en cualquier año mientras que en el plot de áreas el eje Y mide el total de películas de todas las categorías. Esta diferencia es realmente fundamental para entender la diferencia entre ambos plots.

Igual que como hicimos en el gráfico de barras podemos forzar que un gráfico de área abarque todo el eje Y en cuyo caso por cada categoría tendremos un porcentaje en lugar de un total. A este tipo de plots se los suele llamar *stream-graphs*

```

1 ggplot(mtypes, aes(x=year, y=total, fill=category))+
  geom_area(position="fill")+
  xlab("Year")+
  ylab("Number of Movies")+
  ggtitle("Movie Categories Across Time")

```

Listing 10.29: Streamgraph para películas

Este plot pierde la información sobre el total de películas en cada año, pero favorece la comparación del porcentaje de películas de cada categoría. Podemos ver que efectivamente hay mas documentales en los últimos años y podemos ver

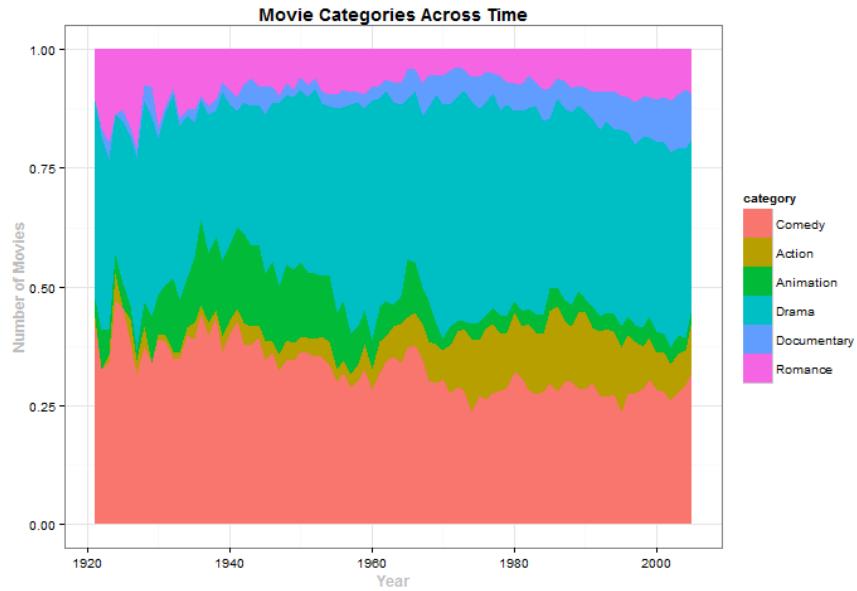


Figure 10.30: Streamgraph para películas

el auge de las películas de animación entre 1935 y 1960 aproximadamente en esos años las películas animadas eran muchas mas que las de acción.

10.9 Heatmaps

Supongamos que tenemos un set de datos sobre jugadores de la NBA de la forma: (name,variable,value) por ejemplo ("Stephen Curry","G",214) indicando que Stephen Curry jugó 214 partidos (G=games). Un heatmap representa en el eje Y todos los puntos, instancias (jugadores) y en el eje X cada una de las categorías posibles. Los ejes suelen ser intercambiables sin que afecte la visualización. El heatmap es entonces una matriz en donde cada celda muestra el valor que toma la variable del eje X para el punto del eje Y.

```

1 ggplot(nba.m, aes(variable , Name)) +
  geom_tile(aes(fill = rescale), colour = "white") +
  3 scale_fill_gradient(low = "white",high = "steelblue")+
  theme(axis.text.x = element_text(angle = 90, hjust = 1))

```

Listing 10.30: Heatmap para la NBA

En el plot podemos ver con valores mas oscuros los valores mas altos, es conveniente re-escalar todos los valores al rango 0-1 para que la visualización sea uniforme.

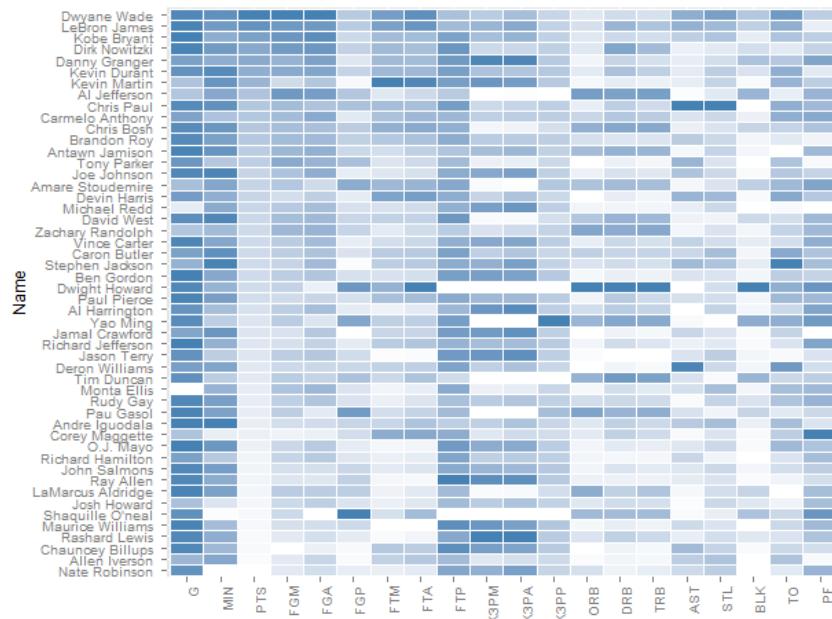


Figure 10.31: Heatmap para la NBA

Buscando los puntos oscuros o claros en cada columna encontramos los jugadores que se destacan ya sea de forma positiva o forma negativa para la categoría en cuestión, mientras que una fila con muchos valores oscuros corresponde a un jugador muy valioso y una fila con valores en su mayoría claros correspondería a un jugador con malas estadísticas.

El color en un heatmap siempre es un gradiente y es convención usar colores oscuros para los valores más altos de un gradiente y colores claros para los valores más bajos. Es muy confuso cuando se nos presenta una visualización con esto invertido. La cantidad de valores en el eje Y puede ser arbitrariamente grande, la cantidad de valores en el eje X debería ser un número manejable para que tenga sentido analizar para una determinada fila todos sus valores. En caso contrario caemos en el problema de no poder buscar para un cierto dato y una cierta variable cuál es su valor.

Una forma de mejorar un heatmap es agrupando los nombres de los datos (jugadores) de acuerdo a su posición y podríamos también agrupar las columnas en estadísticas que tenga sentido comparar columna a columna.

Podemos hacer un heatmap para nuestro set de datos mtcars para comparar los distintos autos:

```
2 mtcars$car=row.names(mtcars)
  cars=melt(mtcars,id="car")
```

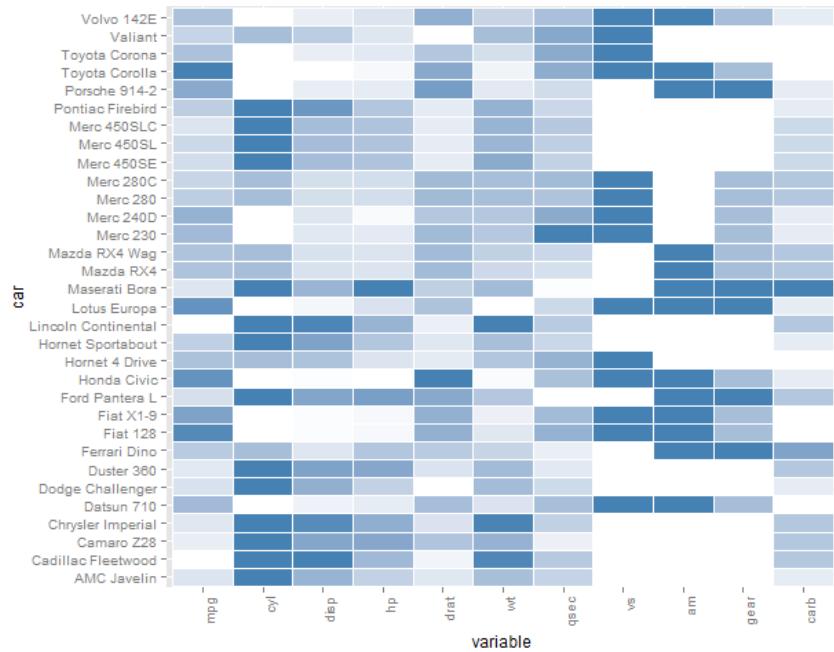


Figure 10.32: Heatmap para MtCars

```

1 cars <- ddply(cars, .(variable), transform, rescale = rescale(value))
2   )
3 ggplot(cars, aes(variable, car)) +
4   geom_tile(aes(fill = rescale), colour = "white") +
5   scale_fill_gradient(low = "white", high = "steelblue")+
6   theme(axis.text.x = element_text(angle = 90, hjust = 1))

```

Listing 10.31: Heatmap para MtCars

En este plot podemos ver que el "outlier" para mpg y con 8 cilindros es el Lincoln Continental que consume muchísimo. Este plot es muy útil para comparar los distintos vehículos considerando todas sus características al mismo tiempo. Por ejemplo podemos decidir comprar un Maserati Bora...

10.10 Radar Charts

Un Radar Chart es un plot en el cual podemos visualizar un pequeño número de instancias en varias dimensiones al mismo tiempo y compararlas. Por ejemplo la figura 10.33 nos muestra un Radar Chart en donde comparamos a tres candidatos a obtener un empleo en 6 dimensiones diferentes: Educación, Experiencia, Comunicación, Conocimiento y Presentación.

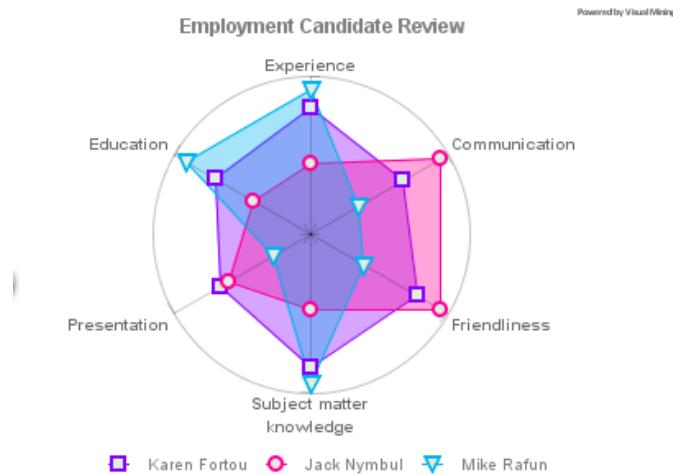


Figure 10.33: Radar Chart

Analizando el chart podemos ver varias cosas, como hay solo tres comparaciones posibles analicemos cada caso. Caso 1: Karen (violeta) vs Jack (rojo). Karen es mejor en Experiencia, Educación y Conocimiento pero Jack es mejor en Comunicación y Friendliness. Segundo parecer Jack es un tipo agradable pero Karen es mejor candidata. Comparemos entonces a Karen con Mike. Mike tiene algo mas de experiencia y algo mas de educación que Karen, pero pierde en Presentación, Comunicación y Friendliness. Aparentemente Mike es Nerd. De todo esto podemos concluir que Karen es la mejor candidata y la superficie que corresponde al color violeta es bastante mayor a las superficies azules y rojas.

10.11 Coordenadas Paralelas

Un plot de coordenadas paralelas también sirve para representar datos en muchas dimensiones pero tiene un fin diferente al radar chart. En un plot de coordenadas paralelas tendremos una línea por cada instancia (dato) y columnas representando cada dimensión, por cada dato trazamos una línea que pase por los valores que toma el dato en cada dimensión.

Este plot sirve para ver tendencias globales en los datos como por ejemplo que los autos que tienen poca potencia tienen valores de aceleración malos, esto lo vemos entre las columnas 4 y 5. También vemos la relación entre la aceleración y el consumo que es un poco menos clara pero muestra que los autos rápidos suelen consumir más.

En un plot de coordenadas paralelas es importante decidir en qué orden vamos a mostrar las diferentes dimensiones de forma tal que las dimensiones

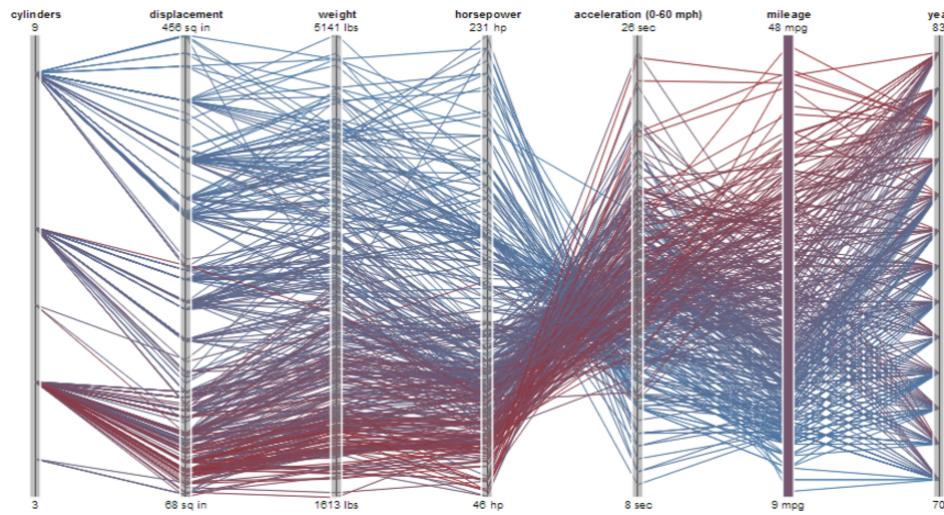


Figure 10.34: Coordenadas Paralelas

para las cuales la forma en que fluyen los datos nos interesa queden como vecinas. Podemos usar el color de la línea para representar una variable categórica por ejemplo en azul los autos manuales y en rojo los automáticos. Así vemos que los autos automáticos tienden a ser mas livianos y tener menos potencia que los manuales. Pero en cuanto al año de fabricación los automáticos son mas nuevos por lo que marcarían una tendencia.

10.12 Treemaps

Un *treemap* sirve para representar información jerárquica, en general comenzamos con un rectángulo que dividimos en sub-rectángulos cuya área es proporcional al valor de una cierta variable, cada uno de estos puede luego subdividirse de acuerdo a otra variable y así sucesivamente. Puede agregarse una dimensión extra a los treemaps mediante el color de los rectángulos.

En el ejemplo vemos un treemap para las exportaciones de Italia, la primera división por colores es por categoría de exportación, dentro de esto mostramos los productos y dentro sub-productos.

Los treemaps se usan, por ejemplo, para mostrar el uso de un disco en una computadora, podemos dividir el rectángulo principal de acuerdo al espacio que consumen las diferentes carpetas y luego por cada carpeta subdividir en subcarpetas y archivos. De esta forma podemos identificar rápidamente cuáles son las carpetas y archivos que mas disco ocupan en caso de que necesitemos liberar espacio.

Los treemaps se usan mucho para información demográfica lo cual resulta

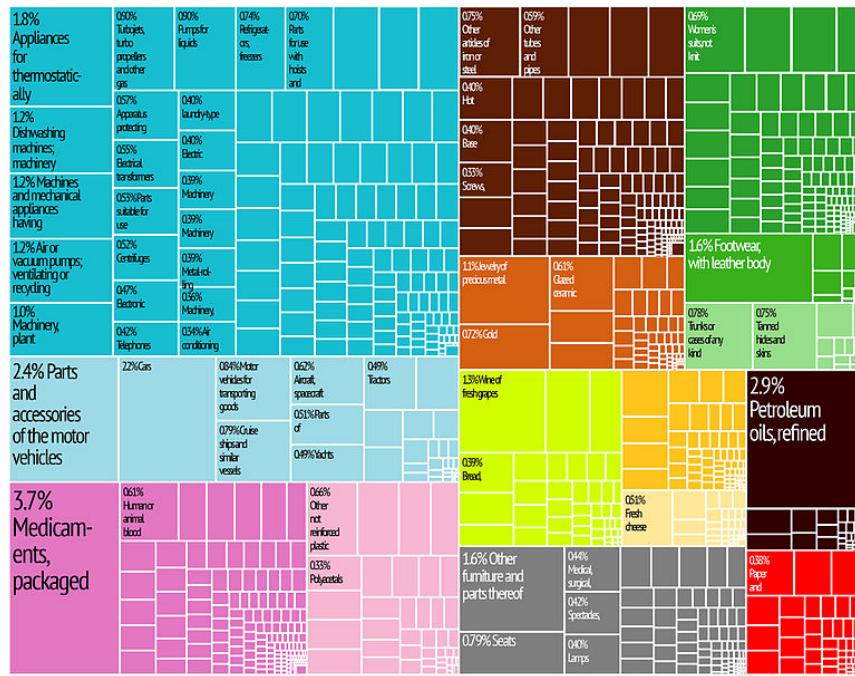


Figure 10.35: Treemaps

evidente porque podemos dividir un país en estados, luego cada estado en condados y si queremos podemos subdividir cada condado en diferentes municipios. De esta forma podemos ver el valor de una variable con diferentes escalas al mismo tiempo.

10.13 Mapas y Choropleths

Un choropleth es un mapa en el cual se muestra el valor de una determinada variable de forma georeferenciada, el mapa se divide en áreas y para cada área se calcula el valor de la variable de interés.

En el ejemplo vemos un mapa sobre la tasa de desempleo en USA, podemos ver que es mayor en la costa Oeste y dentro de cada estado pueden verse las variaciones para los distintos condados.

En un choropleth es muy importante determinar la granularidad de la información es decir en que polígonos dividir el mapa y con que valor para la variable de interés llenar estos polígonos, puede ser el valor máximo, el mínimo, el promedio, la mediana y alguna otra estadística como por ejemplo la desviación standard de la variable. Esto sería importante para evitar ser víctimas de De Moivre y que haya zonas de nuestro mapa con valores inusualmente altos o bajos

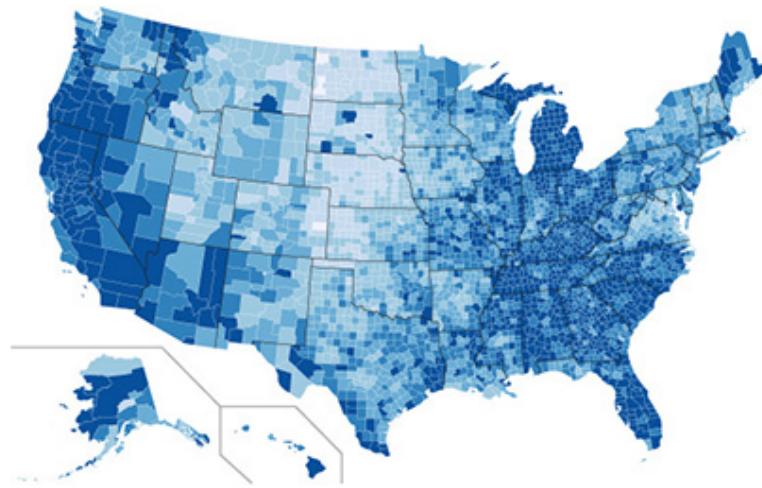


Figure 10.36: Mapa de desempleo en USA

simplemente por tener pocos datos. Lamentablemente la mayoría de los choropleths sufren de este problema y es muy común encontrar que los valores mas claros o oscuros coinciden con las áreas mas despobladas o con menor cantidad de datos.

Chapter 11

Streaming

The final end of Eternity, and the beginning of Infinity

- Isaac Asimov

Un stream es un flujo de datos que no tiene fin. Los streams son muy comunes hoy en día, los paquetes de una red son un claro ejemplo, los posts en Twitter o Facebook, los comentarios en un foro, las ventas en sitios como Amazon o Ebay, todos estos casos representan flujos de datos continuos.

Los algoritmos tradicionales tienen graves problemas al procesar streams. En primer lugar es muy difícil calcular estadísticas, rankings y otros datos interesantes si los datos cambian constantemente, cuando calculamos un cierto valor ya ha perdido actualidad. Por otro lado prácticamente ningún algoritmo puede procesar una cantidad de datos infinitos. Los algoritmos de streaming están diseñados para solucionar estos problemas, son algoritmos que trabajan en memoria y procesan los datos a medida que estos ocurren actualizando constantemente su resultado. Esto permite obtener en tiempo real datos sobre un determinado stream.

Imaginemos que somos Amazon, la cantidad de estadísticas y datos que pueden interesarnos son enormes. Por ejemplo: ¿Cuáles son los productos más vendidos en las últimas horas? ¿Cuántos productos diferentes se han vendido? ¿Hay algún producto que se esté vendiendo más de lo habitual?, ¿Cuántos Iphones vendimos en las últimas 24 horas?, etc. Todas estas preguntas son prácticamente imposibles de resolver sin usar algoritmos de streaming por el simple hecho de que en 2015 Amazon generaba U\$S 120.000 en ventas cada... 5 segundos(!)

11.1 Reservoir Sampling

Nuestro primer algoritmo es muy importante porque permite convertir un stream infinito en un conjunto de datos finito, la idea es tener en cada momento una

muestra (*sample*) del stream en memoria y calcular los datos y estadísticas que nos interesen sobre el stream a partir de la muestra usando algoritmos tradicionales. El algoritmo necesita, entonces, mantener en memoria una cantidad fija de elementos k , para que las estadísticas tengan sentido debemos garantizar que la probabilidad de que un dato del stream esté en la muestra sea igual a:

$$P(X_i \in \text{sample}) = \frac{k}{N} \quad (11.1)$$

Donde N es la cantidad de elementos del stream hasta el momento. Notemos que la probabilidad es dinámica es decir que a medida que observamos mas datos la probabilidad de que un dato esté en la muestra debe ser menor y, lo que es mas importante, la probabilidad de todos los elementos independientemente de cuando los observamos tiene que ser igual.

El algoritmo funciona de la siguiente manera: por cada dato que genera nuestro stream calculamos $p = k/n$ que es la probabilidad de que el dato ingrese a la muestra, luego generamos un número random y de acuerdo a p decidimos si el dato ingresa o no a la muestra, por ejemplo si $p = 0.6$ generamos un random entre 0 y 1 y el dato ingresa a la muestra si el número es menor a 0.6. Si el dato no ingresa la muestra no cambia, si el dato ingresa entonces elegimos al azar un dato de la muestra y lo reemplazamos por nuestro dato.

Consideremos el siguiente ejemplo: $k = 3$ y nuestro stream consiste en números y observamos: 1,6,2,3. Los primeros k (tres) números ingresan a la muestra, por lo que tendremos en la muestra 1,6,2. La probabilidad de que 3 ingrese es ahora $3/4$. La probabilidad de que 1 siga en la muestra se puede calcular como la probabilidad de que 3 no ingrese ($1/4$) mas la probabilidad de que 3 ingrese pero 1 no sea reemplazado es decir $(3/4 * 2/3 = 1/2)$. La probabilidad del 1 es entonces $1/4 + 1/2 = 3/4$, de esta forma vemos que con 4 elementos la probabilidad de cada uno de ellos es igual a $3/4$ que es lo que queríamos.

Theorem 15 (Reservoir Sampling) *Dado un stream de n elementos la probabilidad de que un elemento esté en la muestra es k/n*

Vamos a demostrar esto por inducción.

Suponemos que luego de n elemento la probabilidad es k/n queremos ver que con $n+1$ elementos la probabilidad es $k/(n+1)$

Caso base: si $n = k$ entonces la probabilidad de cada elemento es $k/k = 1$.

Cuando el elemento $n+1$ es observado podemos calcular la probabilidad de que un elemento permanezca en la muestra como:

probabilidad de que $n+1$ no ingrese + probabilidad de que $n+1$ ingrese por la probabilidad de que el elemento no sea eliminado.

probabilidad de que $n+1$ no ingrese = $1 - (k/(n+1))$ probabilidad de que $n+1$ ingrese = $k/(n+1)$ probabilidad de que el elemento sea reemplazado: $k - (1/k)$

$$1 - \left(\frac{k}{n+1}\right) + \left(\frac{k}{n+1}\right)\left(k - \frac{1}{k}\right) = \frac{n}{(n+1)}$$

Entonces en el paso n el dato está en la muestra con probabilidad k/n y en el paso $n+1$ el dato está en la muestra con probabilidad $n/(n+1)$.

Por lo tanto la probabilidad de que un dato esté en la muestra en el paso $n + 1$ es la probabilidad de que esté en el paso n por la probabilidad de que se mantenga en el paso $n + 1$

$$\left(\frac{k}{n}\right)\left(\frac{n}{n+1}\right) = \frac{k}{n+1}$$

Que es lo que queríamos demostrar.

El algoritmo de Reservoir sampling es interesante porque nos da una forma práctica de resolver el problema de los datos infinitos usando una muestra de tamaño finita representativa de los mismos.

11.2 Momentos de un Stream

Un stream es una colección de elementos, llamemos M_i a la cantidad de veces que el elemento i ocurrió en el stream. Por ejemplo si el stream es 1,2,3,2,4 entonces $M_1 = 1, M_2 = 2, M_3 = 1$ y $M_4 = 1$, observemos que el dato no necesariamente tiene que ser un número, el stream podría ser a,b,c,b,d y los valores serían los mismos: $M_a = 1, M_b = 2, M_c = 1, M_d = 1$. Vamos a definir el momento de orden k de un stream como:

$$M^k(S) = \sum_{i \in S} M_i^k \quad (11.2)$$

El momento de orden 0 implica elevar la cantidad de veces que cada elemento distinto del stream aparece a la cero por lo que vamos a contar 1 por cada elemento diferente del stream. Por lo tanto el momento de orden 0 es la cantidad de elementos distintos en el stream. Calcular el momento de orden 0 no es sencillo ya por cada elemento del stream necesitamos saber si es nuevo o repetido porque solo los elementos nuevos suman, pero no podemos tener todos los elementos distintos que hemos observado en memoria y comparar porque estos podrían ser miles de millones y comparar cada elemento contra miles de millones es ineficiente.

El momento de orden 1 es simplemente la cantidad de elementos en total en el stream, es decir n , para calcular este momento no hace falta ningún algoritmo especial ya que simplemente sumamos 1 cada vez que observamos un dato del stream.

El momento de orden 2 del stream, es un poco mas difícil de interpretar y se lo conoce como *número sorpresa* del stream. Es un indicador de si los datos del stream se distribuyen de forma pareja o de si hay un elemento que predomina sobre los demás.

Por ejemplo sea $S_1 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1)$, $S_2 = (1, 1, 2, 3, 1, 1, 4, 5, 1, 7)$

$$M^0(S_1) = 9 \quad M^1(S_1) = 10 \quad M^2(S_1) = 12$$

$$M^0(S_2) = 6 \quad M^1(S_2) = 10 \quad M^2(S_2) = 30$$

Para calcular el momento de orden 0 de un stream veremos el algoritmo de Flajolet-Martin y un derivado del mismo conocido como HyperLogLog.

11.3 Flajolet-Martin y el HyperLogLog

11.3.1 Flajolet-Martin

El algoritmo de [Flajolet-Martin] es sumamente simple y sorprendente, implica solamente el uso de una función de hashing y un contador en memoria. Por cada dato que observamos en nuestro stream aplicamos la función de hashing y contamos la cantidad de 0s consecutivos en la misma a partir del bit 0. En memoria llevaremos un contador r con la cantidad máxima de ceros que hemos observado al comienzo de cualquier resultado de la función de hashing.

El momento de orden 0 del stream se estima como:

$$M^0(S) \approx 2^r \quad (11.3)$$

Ejemplo:

$S=a, b, a, c, b, c, d$

```

h(a) = 01000101 r=1
h(b) = 10010111 r=1
h(a) = 01000101 r=1
h(c) = 00101101 r=2
h(b) = 10010111 r=2
h(c) = 00101101 r=2
h(d) = 01010011 r=2

```

$M_0 = 2^2 = 4$

La intuición detrás del algoritmo es la siguiente. Si la función de hashing es buena entonces un 50% de los hashes comienzan con 0, un 25% de los hashes comienzan con 00, un 12.5% de los hashes comienzan con 000, por lo tanto calcular 2^r aproxima a la cantidad de elementos que vimos en nuestro stream. Visto de otra forma si hemos visto 2^r elementos diferentes en el stream entonces alguno de ellos tiene que comenzar con r ceros (si no hay colisiones).

Como en el mundo real las colisiones existen el algoritmo no es perfecto y es muy sensible a los valores de la función de hashing. Por ejemplo si para algún dato la función de hashing nos diera 0 entonces todo el algoritmo dejaría de funcionar. Una forma de solucionar esto es usando k funciones de hashing al mismo tiempo y llevando k contadores r_i en memoria.

Si tenemos k estimadores debemos notar que el promedio no es una buena estimación porque es muy sensible a valores extremos en alguno de los estimadores. La mediana es mas adecuada. El problema de la mediana es que siempre será una potencia de 2 y esto no es bueno ya que solo podríamos estimar potencias de 2 como resultado del algoritmo.

La solución en el algoritmo de Flajolet-Martin pasa por dividir los k estimadores en b grupos de m estimadores cada uno y calcular como resultado final el promedio de las medias de cada grupo.

Ejemplo:

$$r_1 = 2 \ r_2 = 5 \ r_3 = 1 \ r_4 = 3 \ r_5 = 3 \ r_6 = 4$$

Si usamos 2 grupos de 3 estimadores cada uno entonces en el primer grupo tenemos 4,32,2 y la mediana es 4. En el segundo grupo tenemos 8,8 y 16 y la mediana es 8. Por lo tanto nuestra estimación es el promedio entre 8 y 4 es decir 6. Vamos a estimar que nuestro stream tiene 6 elementos distintos.

11.3.2 HyperLogLog

HyperLogLog (HLL) [Flajolet] es una versión mejorada de los mismos autores y que es la que hoy en día mas se usa para estimar la cantidad de elementos distintos en un stream. HLL usa una única función de hashing, por ejemplo Jenkins y sobre la misma construye varios estimadores, calculando el resultado final como el promedio armónico de los estimadores. Eso es todo!

La función de hashing nos genera un número de 64 bits, los primeros k bits los vamos a usar para el número de estimador. Por ejemplo si queremos 32 estimadores usamos los primeros 5 bits para calcular el número de estimador. De los restantes bits contamos como siempre la cantidad de ceros al comienzo de los mismos y actualizamos el estimador si esta cantidad es mayor a la anterior.

$$M^0(S) \approx \frac{n}{\sum_{j=1}^n 2^{-R_j}} \quad (11.4)$$

Por ejemplo si tenemos los siguientes estimadores: 3,5,2,2,3 el promedio es $(8 + 32 + 4 + 4 + 8)/5 = 11.2$ mientras que el promedio armónico es $5/(1/8 + 1/32 + 1/4 + 1/4 + 1/8) = 6.4$. El promedio armónico mitiga el efecto de valores muy grandes en el promedio y aumenta el efecto de valores pequeños lo cual es ideal para el algoritmo HLL.

11.4 AMS

El algoritmo AMS[Alon-Matias-Szegedy] sirve para estimar el momento de orden 2 de un stream, es decir su número sorpresa. Este algoritmo mantiene en memoria k estimadores, cada estimador registra un elemento del stream y la cantidad de veces que apareció el mismo. Por cada elemento del stream si el mismo está entre los estimadores simplemente aumentamos el contador en 1, si el elemento no está entre los estimadores entonces ingresa a memoria con probabilidad k/n igual que en reservoir sampling si el estimador ingresa a la memoria reemplaza a otro estimador elegido al azar.

Cada estimador estima el modelo de orden 2 del stream mediante:

$$M^2(S) \approx n(2 * C_i - 1) \quad (11.5)$$

Siendo C_i el contador y n la cantidad de elementos del stream. Al igual que en Flajolet-Martin vamos a agrupar los estimadores en b grupos de m estimadores cada uno siendo el resultado final la mediana del promedio de cada

grupo. Notar que esto es al revés que en FM porque ahora los estimadores no son potencia de 2.

Ejemplo: Usamos 3 estimadores ($k = 3$) y nuestro stream es 1,2,3,2,4,2,5,3,4,4,3,1. Vamos a indicar con un * si el estimador ingresa a memoria es decir si salió favorecido en el "sorteo" que hacemos para determinar si le toca ingresar.

```
Dato = 1 k1.elem=1 k1.val=1
Dato = 2 k1.elem=1 k1.val=1; k2.elem=2 k2.val=1
Dato = 3 k1.elem=1 k1.val=1; k2.elem=2 k2.val=1;k3.elem=3 k3.val=1
Dato = 2 k1.elem=1 k1.val=1; k2.elem=2 k2.val=2;k3.elem=3 k3.val=1
Dato = 4 k1.elem=1 k1.val=1; k2.elem=2 k2.val=2;k3.elem=3 k3.val=1
Dato = 2 k1.elem=1 k1.val=1; k2.elem=2 k2.val=3;k3.elem=3 k3.val=1
Dato = 5*k1.elem=1 k1.val=1; k2.elem=2 k2.val=3;k3.elem=5 k3.val=1
Dato = 3 k1.elem=1 k1.val=1; k2.elem=2 k2.val=3;k3.elem=5 k3.val=1
Dato = 4*k1.elem=4 k1.val=1; k2.elem=2 k2.val=3;k3.elem=5 k3.val=1
Dato = 4 k1.elem=4 k1.val=2; k2.elem=2 k2.val=3;k3.elem=5 k3.val=1
Dato = 3 k1.elem=4 k1.val=2; k2.elem=2 k2.val=3;k3.elem=5 k3.val=1
Dato = 1 k1.elem=4 k1.val=2; k2.elem=2 k2.val=3;k3.elem=5 k3.val=1
```

El número sorpresa real del stream es: $2^2 + 3^2 + 3^2 + 3^2 + 1^2 = 32$

La estimación para $k1$ es $12(2 * 1 - 1) = 36$ La estimación para $k2$ es $12(2 * 3 - 1) = 60$ La estimación para $k3$ es $12(2 * 1 - 1) = 12$

Asumiendo que usamos 3 grupos de 1 estimador cada uno la mediana es 36 que se acerca bastante al valor real para el momento de orden 2 del stream. En este caso como cada grupo tiene solo un estimador.

11.5 DGIM

Para el siguiente problema vamos a considerar Streams binarios, es decir que cada dato puede ser 1 o 0. Tomamos una ventana fija de m bits con m realmente muy grande y queremos calcular cuántos bits en 1 vimos en los últimos k bits con $k \leq m$

Este modelo es muy flexible y puede usarse en muchísimas situaciones. Por ejemplo podemos tener un stream por producto y agregar un bit por cada venta realizada indicando con 1 si el vendimos ese producto y con 0 si fue otro. De esta forma podríamos saber cuantas veces se vendió un cierto producto en las últimas 100.000 ventas o cuántas veces se usó un cierto término en Google en las últimas n búsquedas. Etc.

Para comprender que el algoritmo no es trivial debemos entender que m puede ser un número muy grande y necesitamos poder calcular la cantidad de unos en cualquier ventana de tamaño k con $k \leq m$. Contar cuantos unos tenemos por cada consulta no es entonces una opción viable.

El algoritmo que usaremos fue desarrollado por Datar, Gionis, Indyk y Motwani [Datar-Gionis-Indyk-Motwani]. A partir de ahora lo llamaremos DGIM. Es un algoritmo cuyo funcionamiento es realmente muy extraño y de todos los que presentaremos en el capítulo es el mas complejo de entender sin embargo

una vez entendido es un algoritmo bastante simple. Podríamos decir que es algo así como andar en bicicleta, aprender parece difícil pero una vez que lo hacemos no nos olvidamos mas.

Sobre la ventana de m bits el algoritmo va a mantener k sub-ventanas que no pueden superponerse. Por cada sub-ventana va a llevar 2 valores: la posición en la ventana donde comienza la sub-ventana y la cantidad de unos que hay en la misma. Por cada bit procesado a la posición de cada sub-ventana le sumamos uno (para mantener el offset).

Se sabe donde termina cada sub-ventana porque se sabe donde comienza la anterior y la última sub-ventana esta acotada por el fin de la ventana grande: m . La posición donde comienza la ventana se calcula módulo m . En el apunte no lo haremos para no confundir.

Para ver como se construyen estas sub-ventanas veamos como funciona el algoritmo.

Procesamos bit por bit el stream, si el bit es 0 lo ignoramos completamente. Si el bit es 1 entonces creamos una ventana de tamaño 1 para ese bit con contador = 1.

Luego debemos analizar cuántas sub-ventanas hay con 1 bit en total.

Si hay 1 o 2 no hacemos nada.

Si hay 3 entonces combinamos las dos sub-ventanas mas viejas en una nueva sub-ventana de 2 bits.

Y luego vemos cuantas sub-ventanas de 2 bits hay

Si hay 1 o 2 no hacemos nada.

Si hay 3 entonces combinamos las dos sub-ventanas mas viejas en una nueva sub-ventana de 4 bits.

Y asi sucesivamente...

Ejemplo, tomando un m que suponemos grande para que no influya.

Notar que el offset de las sub-ventanas siempre apunta al bit mas reciente de las mismas y que ese bit siempre es 1. Recordar que siempre se combinan las sub-ventanas mas viejas.

```

Stream: [1]
[pos=0 bits=1]
Stream: [1]0
[pos=1 bits=1]
Stream: [1]0[1]
[pos=2 bits=1] [pos=0 bits=1]
Stream: [1]0[1]
[pos=3 bits=1] [pos=1 bits=1]
Stream: [1]0[1]00
[pos=4 bits=1] [pos=2 bits=1]
Stream: [1]0[1]00[1]
[pos=5 bits=1] [pos=3 bits=1] [pos=0 bits=1]
Aqui tenemos 3 sub-ventanas con 1 bit, combinamos las dos mas viejas
Stream: [101]00[1]
[pos=3 bits=2] [pos=0 bits=1]
```

```

Stream: [101]00[1][1]
[pos=4 bits=2] [pos=1 bits=1] [pos=0 bits=1]
Stream: [101]00[1][1]0
[pos=5 bits=2] [pos=2 bits=1] [pos=1 bits=1]
Stream: [101]00[1][1]0[1] (combinar)
Stream: [101]00[11]0[1]
[pos=6 bits=2] [pos=2 bits=2] [pos=0 bits=1]
Stream: [101]00[11]0[1][1]
[pos=7 bits=2] [pos=3 bits=2] [pos=1 bits=1] [pos=0 bits=1]
Stream: [101]00[11]0[1][1]0
[pos=8 bits=2] [pos=4 bits=2] [pos=2 bits=1] [pos=1 bits=1]
Stream: [101]00[11]0[1][1]0[1] (combinar)
Stream: [101]00[11]0[11]0[1] (combinar, hay 3 de longitud 2)
Stream: [1010011]0[11]0[1]
[pos=5 bits=4] [pos=2 bits=2] [pos=0 bits=1]

```

Con la estructura de sub-ventanas para calcular cuantos 1s hay en una cierta sub-ventana de longitud k lo que hacemos es sumar la cantidad de bits de todas las sub-ventanas comprendidas y la mitad de la ultima.

Ejemplo si tenemos la salida de nuestro seguimiento:

```
[pos=5 bits=4] [pos=2 bits=2] [pos=0 bits=1]
```

Y queremos saber cuantos 1s hay en los últimos 6 bits del stream notamos que las ventanas de pos=3 y pos=0 quedan dentro de estos 6 bits y la ventana de pos=11 parcialmente. Entonces estimamos $1 + 2 + 4/2 = 5$ bits

Si recordamos el stream: 101001101101

Vemos que la cantidad real de 1s en los ultimos 6 bits era 4, la estimación es cercana.

Sin embargo para este caso particular como sabemos que buscamos 6 bits y la tercera sub-ventana justo comienza en el sexto bit podemos saber que ese bit es un uno pues todas las ventanas comienzan con 1. Por lo tanto podemos calcular $1 + 2 + 1 = 4$ bits. (es un caso particular).

DGIM necesita en promedio $\log(2M)$ sub-ventanas y por cada sub-ventana necesita $\log N$ bits para la posición y $\log \log N$ bits para la cantidad de 1s. Es decir que el espacio en memoria siempre está acotado al logaritmo del tamaño de la ventana y esto es perfectamente manejable.

El error de DGIM es de aproximadamente un 50%, esto es bastante alto. Para reducirlo lo que hacemos es en lugar de admitir 2 sub-ventanas como máximo de cada tamaño admitir " r ". Haciendo esto el error de DGIM es de $1/r$. Es decir que si queremos un error del 10% admitimos hasta 10 ventanas de cada longitud pero necesitamos almacenar mayor cantidad de sub-ventanas con mayor consumo de memoria.

11.5.1 Extensión para valores enteros

Supongamos que el stream en lugar de ser binario es de valores enteros y queremos la suma de los últimos k elementos con $k < m$ siendo m un número muy grande.

Se puede usar DGIM fácilmente para resolver este problema si suponemos que los enteros son números de "z" bits cada uno. Podemos entonces usar un DGIM por cada bit del número.

Luego tenemos la cantidad 1s en promedio en los últimos k elementos por cada bit del número y podemos estimar el número como

$$\sum C_i * 2^i \quad (11.6)$$

11.6 Decaying Windows

Otra forma de analizar el problema de contar cuantos 1s aparecen en los últimos bits de un stream consiste en aplicar una función que vaya decayendo el peso de cada bit en 1 ya observado a medida que procesamos el stream.

Si nos interesan los últimos "n" bits del stream entonces usamos una constante $c = 10^{-n}$, llevamos un contador que comienza en 0 y por cada bit del stream multiplicamos el contador por $1 - c$ y luego sumamos el bit (1 o 0). Ejemplo:

```
Sea c=0.1 (1-c=0.9)
1 (1)
10 (0.9)
100 (0.81)
1001 (1.729)
10011 (2.5561)
100110 (2.30049)
1001100 (2.070441)
10011001 (2.8633969)
etc..
```

En general se usan valores del orden de 10^{-n} con n grande.

Supongamos ahora que tenemos un stream de URLs clickeadas por usuarios y queremos saber cuáles son las URLs mas populares en los últimos "n" clicks. Podemos implementar esto con el algoritmo que presentamos si asociamos a cada URL un stream binario en donde 0 indica que la URL no fue clickeada y 1 que si.

La cantidad de URLs es muy grande por lo que queremos llevar solo una cantidad fija en memoria.

El algoritmo entonces seria algo asi:

Por cada URL del stream

Multiplicar todos los valores en memoria por (1-c)

Eliminar de memoria todas las URLs por debajo de un cierto umbral (ej 0.5)

Si la URL que estamos procesando esta en memoria sumarle 1 sino agregarla con valor 1.

Por ejemplo sea el stream ABCABBABCD Y sea $c = 0.1$ con umbral 0.6

	A	B	C	D
A	1			
B	0.9	1		
C	0.81	0.9	1	
A	1.73	0.81	0.9	
B	1.56	1.73	0.81	
B	1.4	2.56	0.73	
A	2.26	2.3	0.66	
B	2.03	3.07		
C	1.83	2.76	1	
D	1.65	2.48	0.9	1

Table 11.1: Ejemplo Decaying Windows.

11.7 Filtros de Bloom

Dado un stream queremos saber si los elementos que observamos en el mismo pertenecen o no a un cierto conjunto de elementos predefinidos. Por ejemplo podemos tener una base de direcciones de email confiables y queremos analizar si un email puede ser spam o no verificando si su emisor está o no en esta lista de direcciones confiables.

La "base" de elementos contra los cuales verificamos es siempre muy grande, no sirve mantenerlas en memoria en un hash o estructura similar.

El método de filtrado mas popular consiste en usar los llamados *Filtros de Bloom*. Un filtro de Bloom es un vector binario de B bits y k funciones de hashing $0..B$. Para "agregar" un elemento al filtro le aplicamos las funciones de hashing y luego encendemos en 1 los bits apuntados por las funciones. Se prenden k o menos bits según haya o no colisiones.

Para verificar si un dato del stream pertenece a nuestro conjunto le aplicamos las funciones de hashing y verificamos si los bits están en todos en 1. Si alguno está en 0 entonces el elemento no pertenece al conjunto. Si todos los bits están en 1 entonces el elemento pertenece al mismo con una cierta probabilidad ya que podría haber un falso positivo si alguno de esos bits fue encendido por algún otro elemento.

Si tenemos k funciones de hashing, n elementos a hashear y m bits. Luego de hashear todos los elementos. ¿Cuántos bits quedan en 1 y cuántos en 0 en el vector? ¿Cuál es la probabilidad de un falso positivo?

Comenzamos con algunos cálculos básicos:

La probabilidad de que una función encienda un cierto bit es $1/m$

La probabilidad de que una función no encienda un cierto bit es $1 - (1/m)$

Si tenemos k funciones de hashing la probabilidad de que ninguna encienda un cierto bit es:

$$\left[1 - \left(1/m\right)\right]^k$$

Luego de insertar n elementos la probabilidad de que un cierto bit siga siendo 0 es

$$\left[1 - \left(1/m\right)\right]^{kn}$$

La probabilidad de que las k posiciones a testear sean 1 entonces es:

$$(1 - \left[1 - \left(1/m\right)\right]^{kn})^k$$

Y una estimación para esta fórmula es:

$$(1 - e^{kn/m})^k \quad (11.7)$$

Esta función se puede graficar para probar valores de n, m y k que funcionen correctamente.

Para valores de n y m fijos el valor óptimo de k se puede calcular de la forma:

$$k = \frac{m}{n} \log 2 \quad (11.8)$$

Para m y n fijos el gráfico de k según la probabilidad de falsos positivos es:

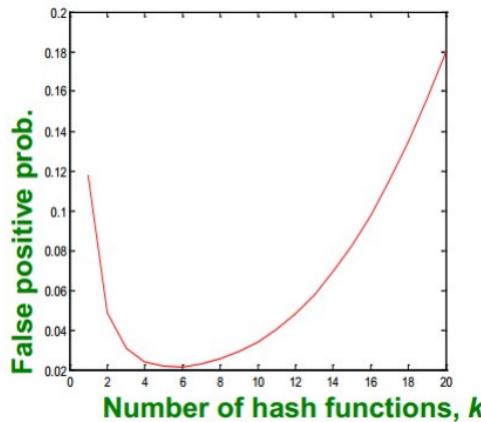


Figure 11.1: Filtros de Bloom FP prob según k

Como podemos ver la probabilidad de un falso positivo va bajando hasta llegar al óptimo y luego sube ya que hay demasiados bits en 1 en el filtro.

Si usamos el k óptimo entonces la cantidad de bits a usar puede calcularse como:

$$m = \frac{n \ln p}{\ln(2)^2} \quad (11.9)$$

Siendo p la probabilidad que queremos para un falso positivo.

Ejemplo:

Si tenemos mil millones de direcciones de email y queremos que la probabilidad de un falso positivo sea: $p = 0.01$ entonces:

$$m = -(1000000000 \ln 0.01) / (\ln 2)^2$$

$m = 9585$ millones de bits

y $k = 2.88$ es decir 3 funciones de hashing

Algo interesante sobre los filtros de Bloom es que puede usarse el filtro para estimar la cantidad de elementos que han sido insertados en el mismo [Swamidass-Baldi]

$$n \approx -\frac{m \ln(1-x/m)}{k} \quad (11.10)$$

M = cantidad de bits del filtro.

X = cantidad de bits en 1.

Los filtros de Bloom son muy sencillos en su concepto pero aplicarlos correctamente requiere el uso de estas fórmulas y no todo el mundo las conoce!

11.7.1 Extensión para admitir borrado

Si queremos poder insertar y eliminar elementos del filtro entonces la estructura normal no sirve porque no podemos al borrar apagar los bits en 1 indicados por las k funciones de hashing pues estos bits podrían corresponder a otros elementos ya insertados en el filtro y luego al buscarlos el filtro nos diría que no están.

La solución es usar un *counting filter* que en lugar de 1 bit por cada posición tiene un entero de f bits, de esta forma cada vez que se inserta se incrementan los enteros apuntados por las funciones de hashing y cuando se elimina se decrementa. La pertenencia es simplemente verificar que las posiciones sean todas distintas de cero.

La desventaja, evidente, es la mayor cantidad de espacio necesaria.

11.8 Count-Min

El algoritmo *Count-Min Sketch* [Muthukrishnan-Cormode] es uno de los algoritmos de streaming mas importantes porque permite para cualquier dato de un stream estimar cuántas veces ocurrió el mismo hasta el momento. Esto permite responder todo tipo de problemas siendo uno de los mas frecuentes el denominado *Heavy Hitters Problem* por lo que empezamos esta sección con un análisis teórico de este problema.

11.8.1 The Heavy Hitters Problem

El problema de los *Heavy Hitters* consiste simplemente en encontrar los elementos más frecuentes (populares) en un stream. Una forma de plantearlo es dado un stream de n elementos, y un cierto valor k , siendo n un número muy grande y k un número relativamente chico encontrar los elementos que ocurren al menos n/k veces en el stream. El caso particular de $k = 2 - \delta$ con δ pequeño consiste en encontrar un elemento que sea mayoría en el stream.

Este problema puede verse también de otra forma: encontrar los k elementos más frecuentes en el stream.

El problema tiene una enorme cantidad de aplicaciones: los productos más vendidos en Amazon, los hashtags más populares en Twitter, las IPs que más paquetes envían a un router, los queries más populares en Google, etc. Es fácil pensar cientos o miles de cosas interesantes que son variantes de nuestro problema.

La solución trivial consiste en ordenar todos elementos del stream $O(n \log n)$ y luego recorrer linealmente el stream contando la cantidad de veces que aparece cada elemento. Esto, como podemos imaginar, es inaceptable en un stream porque simplemente es imposible tener todos sus elementos en memoria y mucho menos ordenarlos.

Vamos a demostrar que lamentablemente no hay forma de resolver el problema de los Heavy Hitters en ninguna de sus variantes en $O(n)$ sin que la complejidad espacial se dispare.

Theorem 16 *No existe algoritmo que resuelva el HH problema en $O(n)$ usando una cantidad sublineal de espacio auxiliar.*

Para demostrar esto vamos a usar una instancia en particular del problema que luego por inducción puede extenderse a cualquier caso. Sea $k = n/2$ es decir que queremos encontrar los elementos que ocurren al menos 2 veces en el stream. Supongamos que nuestro stream S tiene la forma:

$$S = x_1, x_2, x_3, \dots, x_{n-1}, x_n$$

Llamemos $S_{n-1} = x_1, x_2, x_3, \dots, x_{n-1}$ si queremos ver si x_n es un HH en S necesitamos simplemente comprobar si $x_n \in S_{n-1}$. Esto quiere decir que resolver el problema HH se reduce al problema de resolver la pertenencia de un elemento a un conjunto.

Es trivial demostrar que no se puede resolver el problema de pertenencia de un elemento a un conjunto sin almacenar el conjunto. Cualquier dato que descartemos del mismo puede ser el elemento que estamos buscando.

Por lo tanto el problema HH no se puede resolver en $O(n)$ sin almacenar al menos $O(n)$ elemento en memoria. Estas son malas noticias ya que sabemos que no podemos tener el stream almacenado en memoria. Es necesario, si queremos resolver este problema en un stream contar con una solución aproximada ya que hemos visto que la solución exacta es imposible.

El problema de los HH aproximado

En el problema aproximado tenemos un stream S de n elementos y dado un cierto k queremos cumplir dos condiciones:

1. Todo valor que ocurre al menos n/k veces en S está en el resultado
2. Todo valor en el resultado ocurre al menos $\frac{n}{k} - \epsilon n$ veces en S
3. El espacio usado debe ser $O(1/\epsilon)$

Notemos que si ϵ tiende a cero en cuyo caso el problema aproximado se vuelve el problema exacto el costo de almacenamiento $1/\epsilon$ tiende a infinito. Supongamos que usamos $\epsilon = \frac{1}{2k}$, el algoritmo aproximado tiene que emitir cualquier valor con frecuencia al menos $\frac{n}{k}$ pero puede también emitir valores con frecuencia al menos $\frac{n}{2k}$ el espacio usado por el algoritmo debe ser $1/\epsilon = 2k$ que es perfectamente viable por ser k un número manejable.

11.8.2 The Count-Min Sketch

Abordemos ahora el algoritmo que nos interesa: Count-Min. Este algoritmo es bastante simple de explicar, es equivalente a tener ℓ Counting Filters. Necesitamos entonces ℓ funciones de hashing, llamaremos b al espacio de direcciones de las mismas. Por cada dato del stream, aplicamos las ℓ funciones de hashing e incrementamos el bucket apuntado por las funciones en cada uno de los filtros. La estimación del algoritmo sobre la cantidad de veces que apareció un ítem es el mínimo de los ℓ contadores.

Por ejemplo supongamos que tenemos los siguientes filtros ($\ell = 3$):

[4,3,2,6,3,2,1,3]
[5,3,4,3,9,4,9,8]
[3,6,7,4,9,9,5,3]

Y para un cierto elemento las funciones de hashing nos dan $h1=3$, $h2=1$ y $h3=0$. Nuestra estimación para la cantidad de veces que ocurrió este elemento es $\min(6, 3, 3) = 3$. Si el elemento ocurre nuevamente entonces incrementamos los filtros y quedarían:

[4,3,2,7,3,2,1,3]
[5,4,4,3,9,4,9,8]
[4,6,7,4,9,9,5,3]

Usando el algoritmo es fácil resolver el problema de los HH ya que por cada elemento del stream podemos estimar su cardinalidad. El problema de los top- k elementos también es sencillo, solo necesitamos una tabla de k elementos y cada vez que ocurre un elemento en el stream estimamos su cardinalidad y actualizamos el ranking según sea necesario.

Intuitivamente podemos darnos cuenta que Count-Min nunca va a estimar un número menor a la cardinalidad real de un elemento, pero puede estimar un número mayor debido a colisiones. En la próxima sección desarrollaremos la teoría detrás del algoritmo para estimar el error del mismo.

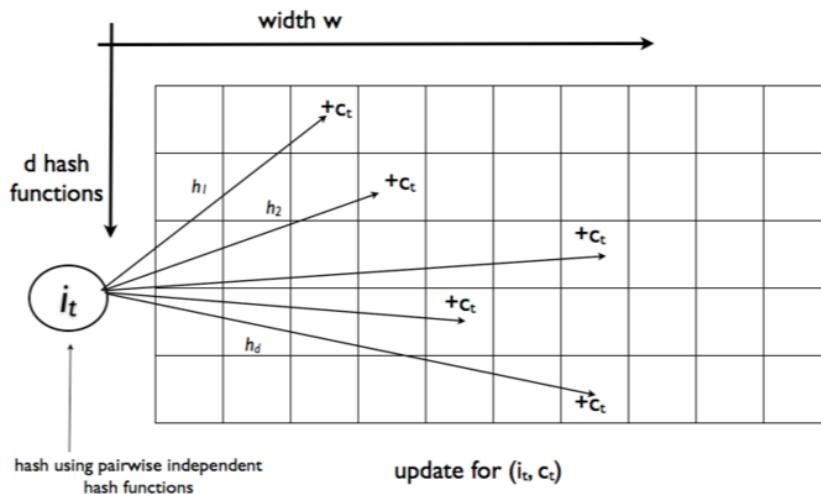


Figure 11.2: Count-Min Sketch

11.8.3 Claves Transistorias: Bit-Marking

Una característica común en muchos streams es que contienen claves *transistorias*, éstas son claves que están activas durante un período de tiempo corto del stream. Por ejemplo en un stream de ventas de productos un producto que se vende durante sólo unos días es una clave transistoria. Un buen ejemplo de esto son los productos navideños.

El problema es que las claves transistorias al colisionar con las claves que podríamos llamar *permanentes* contribuyen a la saturación de Count-Min es decir a que la estimación tenga mayor probabilidad de ser una sobre-estimación de la frecuencia real de algunos ítems. El algoritmo de Bit-Marking [Dimitropoulos] nos puede ayudar a mitigar el problema.

Procesamos el stream en ventanas de tamaño w . Y vamos a usar como estructura adicional de igual dimensión que la estructura de count-min pero con un bit en lugar de un número por cada celda.

Incialmente todos los bits son cero. Cuando uno de los filtros es actualizado encendemos el bit que corresponde para indicar que el filtro estaba activo dentro de la ventana.

Al finalizar cada ventana recorremos el filtro de bits y para los bits que son cero reseteamos el count-min a cero. Es decir que si una posición de count-min no fue actualizada dentro de una ventana la reseteamos a cero bajo la suposición de que había sido activada por una clave transistoria que ha dejado de ocurrir. Las posiciones del filtro que tienen actividad en todas las ventanas nunca se resetean.

Algorithm 19: Bit-Marking

Data: Stream of keys, Window length w , Number of sketches ℓ , Length of sketches m

```

1 counter = 0;
2 for key do
3   for i=1 to  $\ell$  do
4     CM[i][ $h_i(key)$ ] += 1;
5     b[i][ $h_i(key)$ ] = 1;
6     counter+=1;
7     if counter==w then
8       counter == 0;
9       for k=1 to  $\ell$  do
10         for n=1 to m do
11           if b[k][n]==0 then
12             CM[k][n]=0;
13             b[k][n]=0;

```

Bit-Marking puede disminuir notablemente la cantidad de colisiones en un filtro Count-Min lo cual hace que el error del algoritmo disminuya.

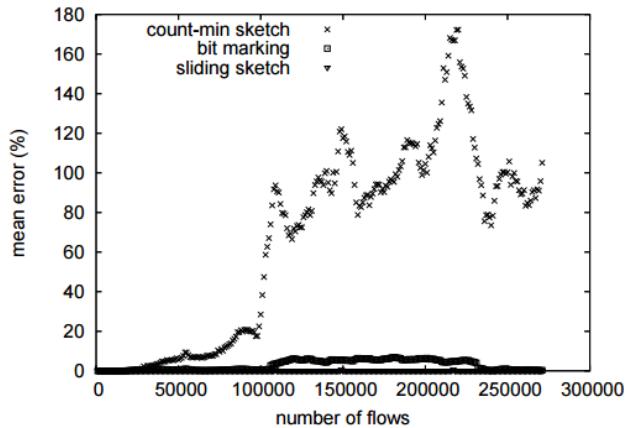


Figure 11.3: Error promedio de Count-Min y error promedio usando Bit-Marking. El Algoritmo Sliding-Sketch es otra variante y tiene un rendimiento casi igual a bitmarking y por eso lo hemos omitido.

Un punto interesante es que podemos usar el algoritmo de bit-marking para detectar eventos especiales en el Stream, reportando los momentos en los cuáles

se eliminan una cantidad importante de claves transitorias. Esto puede indicar que en la o las ventanas inmediatas anteriores sucedió algo que no es habitual en el stream. Solo necesitamos monitorear la cantidad de claves que removemos en cada ventana y fijar un umbral para detectar actividad inusual en el stream.

11.8.4 Teoría de Count-Min

El algoritmo Count-Min soporta únicamente dos operaciones: $\text{Inc}(x)$ y $\text{Count}(x)$. La operación $\text{Count}(x)$ nos debe devolver cuántas veces hemos usado $\text{Inc}(x)$ hasta el momento.

El algoritmo tiene dos parámetros: la cantidad de buckets b y la cantidad de funciones de hashing ℓ . En esta sección vamos a intentar resolver como encontrar estos parámetros, pero por el momento imaginemos que b esta en el orden de los miles y ℓ es un número como 5,6 o 7.

El objetivo del parámetro b es comprimir el stream porque $b \ll n$, esta compresión necesariamente genera errores. El objetivo de ℓ es repetir el experimento varias veces de forma tal de minimizar los errores. Una de las propiedades mas interesantes y sorprendentes de CM es que estos parámetros son **independientes** de la longitud del stream n recordando que n puede ser del orden de billones o trillones.

La estructura de datos de CM es una matriz de $\ell \times b$ que se inicializa en ceros. El procedimiento para $\text{Inc}(x)$ es muy simple:

Algorithm 20: CM Inc

```

1 for i in (1.. $\ell$ ) do
2   CM[i][ $h_i(x)$ ] ++

```

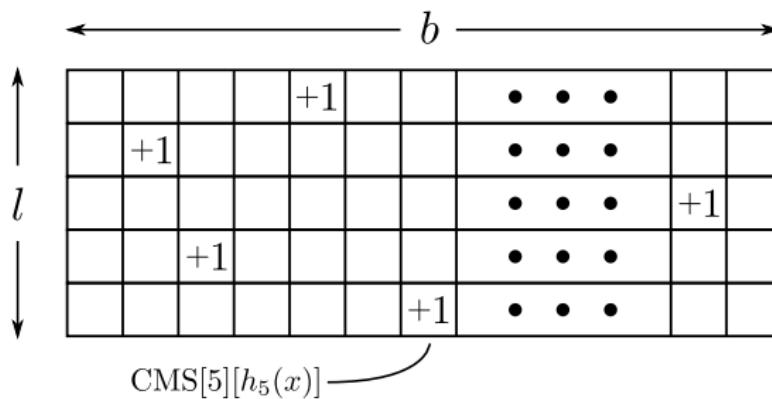


Figure 11.4: Count-Min Sketch

El procedimiento para $\text{Count}(x)$ es muy simple, devolvemos el mínimo de los ℓ contadores:

Algorithm 21: CM Count

```
1 return  $\min_{i=1}^{\ell} \text{CM}[i][h_i(x)]$ 
```

La complejidad de CM es $O(\ell)$ tanto para Inc como para Count.

Como los contadores nunca se decrementan podemos afirmar que $\text{CMS}[i][h_i(x)] \geq f_x$ siendo f_x la frecuencia real de x . Como hemos visto CM nunca puede subestimar f_x pero puede sobre-estimar. Nos interesa saber cuánto puede sobre-estimar el algoritmo. La respuesta depende, por supuesto, de los parámetros b y ℓ . A medida que b crece tenemos menos colisiones y a medida que ℓ aumenta tomamos el mínimo entre mas estimaciones lo cual nos da un resultado mas preciso.

Analicemos primero el caso de un solo filtro individual. Llamaremos Z_i a la estimación que hacemos para un cierto ítem x en el i -ésimo filtro siendo f_x la frecuencia real del ítem. La frecuencia que estimamos usando CM es la suma de la frecuencia real del ítem y y de los elementos que colisionaron con el mismo.

$$Z_i = f_x + \sum_{y \in S} f_y \quad (11.11)$$

Consideremos el siguiente ejemplo: Tenemos un stream con 5 elementos posibles y usamos $b = 3$ sabemos que $h(1) = 2, h(2) = 0, h(3) = 0, h(4) = 1, h(5) = 2$. Para el elemento 4 la estimación de CM va a ser la frecuencia exacta (f_4) ya que es el único elemento que hashea a la posición 1 del filtro. La estimación para el elemento 1 va a ser igual a $f_1 + f_5$ ya que ambos ítems hashean a la misma posición: 2.

Si tomamos las funciones de hashing de una familia universal sabemos que $P[h(x) = h(y)] \leq \frac{1}{b}$ por lo tanto:

$$Z_i \leq f_x + \frac{1}{b} \sum_{y \neq x} f_y \leq f_x + \frac{n}{b} \quad (11.12)$$

Recordemos que en el problema HH aproximado queremos estimar la frecuencia de cada ítem hasta un error de ϵn si usamos $b = \frac{1}{\epsilon}$ y reemplazamos en (9.12) obtenemos:

$$Z_i \leq f_x + \epsilon n \quad (11.13)$$

Esto muestra que el valor de b es independiente de n que es una de las cosas que queríamos demostrar. Intentemos convertir el error en una probabilidad, es decir la probabilidad de que un filtro sobre-estime el valor de f_x .

En esta parte vamos a necesitar la desigualdad de Markov que dice que si X es una variable aleatoria no-negativa y $c > 1$ es una constante entonces:

$$P[X > cE[X]] \leq \frac{1}{c} \quad (11.14)$$

La interpretación de la desigualdad de Markov es sencilla. Supongamos que tenemos una variable aleatoria no-negativa con valor esperado 10. Queremos ver cuántas veces puede tomar un valor mayor a 100 ($c = 10$). En principio es posible que X tome el valor 100 10% de las veces siempre y cuando tome valor 0 el resto de las veces. Pero no puede tomar un valor mayor a 100 mas del 10% de las veces, si así fuera entonces su valor esperado sería necesariamente mayor a 10. Como ejemplo práctico pensemos que si tenemos 10 valores aleatorios de tipo 101,0,0,0,0,0,0,0,0 su valor esperado es $101/10 = 10.1$ que es mas que 10. Si uno de los valores es 100 entonces necesariamente los otros 9 tienen que ser 0 y si un valor es mayor a 100 entonces el valor esperado no puede ser 10 (con 10 elementos). Suponemos que con esto la desigualdad de Markov queda debidamente justificada.

Llamemos ahora X a la sobre-estimación de Count-Min:

$$X = Z_i - f_x \geq 0 \quad (11.15)$$

X no puede ser negativo porque hemos mostrado que CM nunca puede sub-estimar la frecuencia de un elemento. Supongamos que usamos $b = \frac{e}{\epsilon}$ (notar que el numerador es "e" no epsilon). De acuerdo a (9.12) X es a lo sumo $\frac{\epsilon n}{e}$ y podemos aplicar la desigualdad de Markov con valor esperado $E[X] = \frac{\epsilon n}{e}$ y $c = e$ para obtener:

$$P[X > e \frac{\epsilon n}{e}] \leq \frac{1}{e} \quad (11.16)$$

Y entonces:

$$P[Z_i > f_x + \epsilon n] \leq \frac{1}{e} \quad (11.17)$$

Esto es para un solo filtro es decir una sola fila i de nuestra matriz. Recorremos que $\text{Count}(x)$ sobre-estima f_x si y sólo si cada fila sobre-estima f_x en igual forma o mas. Si asumimos que las ℓ funciones de hashing son independientes entonces:

$$P[\min_{i=1}^{\ell} Z_i > f_x + \epsilon n] = \prod_{i=1}^{\ell} P[Z_i > f_x + \epsilon n] \leq \frac{1}{e^{\ell}} \quad (11.18)$$

Entonces si queremos una probabilidad de error δ despejamos ℓ en (9.18) y encontramos que $\ell \geq \ln \frac{1}{\delta}$. Por ejemplo si $\delta = 1\%$ entonces $\ell = 5$ es suficiente. Notemos que para estimar ℓ es decir la cantidad de funciones de hashing solo necesitamos la probabilidad de sobre-estimar que queremos tener, la cantidad de elementos en el stream o el tamaño de los filtros no interesa(!).

Conclusiones obtenidas

- El espacio necesario es por cada filtro $b = \frac{e}{\epsilon} \ln \frac{1}{\delta}$. Para el problema de HH aproximado $\epsilon = \frac{1}{2k}$ es una opción muy común . Para $k = 100$ y $\delta = 0.01$ esto implica unos pocos miles. Para valores mas grandes de k esto escala

linealmente. En todo caso la cantidad es independiente de n (n puede ser billones). Esta es la magia del algoritmo Count-Min podemos descartar prácticamente todos los datos e igual mantener buenas aproximaciones a la frecuencia de cada elemento (!!)

- Las operaciones $\text{Inc}(x)$ y $\text{Count}(x)$ son $O(\ln \frac{1}{\delta})$.
- CM garantiza que el error es siempre una sobre-estimación y nunca una sub-estimación de la frecuencia real del ítem.
- Usando $\ell = \ln \frac{1}{\delta}$ funciones de hashing la probabilidad de sobre-estimar es δ

Desde su publicación en 2004 este algoritmo se ha usado en una gran cantidad de aplicaciones y constituye el estado del arte para la estimación de frecuencias individuales en un stream usando únicamente una pequeña matriz en memoria y procesando los datos uno a la vez, en ciertos casos los problemas que parecen imposibles tienen soluciones realmente muy simples.

Chapter 12

Introducción a Machine Learning

All models are wrong, but some
are useful

- George E.P. Box

Llamamos *Machine Learning* a la rama de la computación que se encarga de construir algoritmos que aprenden a hacer algo útil a partir de los datos. Un ejemplo clásico de ML es el proyecto "Deep mind" de Google en donde un algoritmo procesó durante días videos de Youtube y aprendió a reconocer gatitos. Los algoritmos de ML hoy en día se usan en todo tipo de aplicaciones: por ejemplo en Amazon para recomendarnos que productos comprar, en Twitter para recomendarnos que usuarios seguir, en Google para predecir que páginas son las mas relevantes para una consulta, en un teléfono celular para poder reconocer la voz del usuario, en Facebook para reconocer que fotos contienen rostros de personas, en el mercado inmobiliario para predecir los precios de las propiedades, en el mercado de valores para predecir el precio de una acción. Son tantas las aplicaciones de ML en la industria que se ha producido una verdadera explosión del tema en los últimos años.

Un ejemplo actual y sorprendente de ML es su aplicación a las citas online. En el pasado los sitios que pretendían encontrar la pareja ideal para los usuarios estaban basados en algoritmos que simplemente no funcionaban y su principal objetivo era cobrar el dinero correspondiente a los banners publicitarios y tener el mayor tráfico posible sin preocuparse por la calidad de los resultados obtenidos. Hoy en día sitios como match.com o okcupid usan algoritmos de ML que realmente funcionan y han generado un importante volumen del total de casamientos que se producen por encuentros online.

En el capítulo 1 vimos una introducción a este tema y aprendimos que los algoritmos de ML se dividen en dos grandes ramas: aprendizaje supervisado y aprendizaje no-supervisado. El ejemplo clásico del primero es la clasificación

automática mientras que el segundo se caracteriza por los problemas de clustering. Esta clasificación hace al tipo de problema de ML que queremos resolver, la *forma* en la cual decidimos resolver el problema depende de qué filosofía elegimos y esta depende de a que tribu de la comunidad de ML decidimos confiarle nuestro problema.

12.1 Evaluación de Algoritmos de ML

Casi todos los algoritmos de ML se basan en la construcción de un *modelo* a partir de los datos de forma tal de luego poder usar dicho modelo para predecir datos nuevos. Cada modelo tiene un conjunto de *parámetros* e *hiper-parámetros* que necesita para funcionar. Los parámetros los descubre el algoritmo a partir de los datos, los hiper-parámetros, en cambio, son datos que debemos pasárselos al algoritmo para funcionar. Un ejemplo de hiper-parámetro es el k en KNN.

El problema es cómo encontrar los hiper-parámetros óptimos es decir aquellos que mejor funcionan para nuestro set de datos. Veamos primero un simple esquema basado en el concepto de *validación* del modelo generado.

En general trabajaremos con un *set de entrenamiento* y un *set de test*, la idea es que entrenaremos a nuestro algoritmo con el set de entrenamiento y luego lo aplicaremos al set de test. De esta forma los datos para los cuales queremos probar el algoritmo nunca fueron vistos por el mismo lo cual permite saber si el modelo fue capaz de generalizar correctamente. Para poder validar el modelo necesitamos dividir el set de entrenamiento original en dos: un set de entrenamiento y un set de validación. La idea es entrenar el modelo con el set de entrenamiento y luego probarlo con el set de validación a efectos de encontrar los mejores hiper-parámetros. El set de validación en general es un 20-30% del set de entrenamiento.

Este procedimiento permite probar distintos valores de hiper parámetros mediante grid-search o random-search y elegir los hiper-parámetros que mejor resultado nos den en el set de validación, luego aplicamos el modelo con los hiper parámetros que encontramos al set de test. Es incorrecto usar el set de test como set de validación ya que el algoritmo nunca debe considerar los datos de test para encontrar los valores óptimos para sus hiper-parámetros.

El único problema de este esquema es que siendo el set de validación siempre el mismo podemos caer en el problema de que los hiper-parámetros que encontramos solo sean óptimos para un pequeño conjunto de nuestros datos. Para evitar esto usamos el método de cross-validation que es prácticamente universal para optimizar algoritmos de ML.

12.1.1 Cross Validation

El proceso de *K-fold Cross Validation* comienza partiendo el set de entrenamiento en k bloques, luego vamos a realizar varias iteraciones en las cuales entrenaremos nuestro algoritmo con $k - 1$ bloques y lo validaremos con el restante.

Este proceso se repite k veces para que todos los datos hayan participado alguna vez del set de validación, el resultado es el promedio de las k iteraciones del algoritmo. Esto hay que hacerlo por cada valor posible para nuestros hiperparámetros por lo que dependiendo de los datos puede resultar un proceso costoso.

Un caso extremo de Cross Validation es cuando usamos un único dato como set de validación, en este caso tenemos que hacer n iteraciones del algoritmo, tantas como registros tiene nuestro set de datos. Los resultados son mucho más precisos pero el proceso es, por supuesto, muy costoso y solo tiene sentido con sets de datos reducidos.

12.2 Los Problemas de ML: Overfitting y Underfitting

En esta sección vamos a estudiar los conceptos más importantes en ML: overfitting y underfitting y análogamente bias y variance. Vamos a empezar por intentar explicar la diferencia entre overfitting y underfitting.

El caso más espectacular de overfitting es el caso en el cual usamos KNN con $k = 1$, para el set de entrenamiento KNN es simplemente perfecto porque dado un registro cualquiera el más parecido es el registro mismo y por lo tanto vamos a tener 100% de precisión para el set de entrenamiento. Sin embargo esto no quiere decir que el algoritmo generalice bien y para el set de validación los resultados pueden ser catastróficos.

Otro ejemplo es el ajuste de una función polinómica a un set de datos, si tenemos n puntos en nuestro set de entrenamiento existe un polinomio de grado $n - 1$ que pasa por todos los puntos, sin embargo esto no quiere decir que dicho polinomio sea un buen predictor para puntos nuevos que no estaban en el set de entrenamiento.

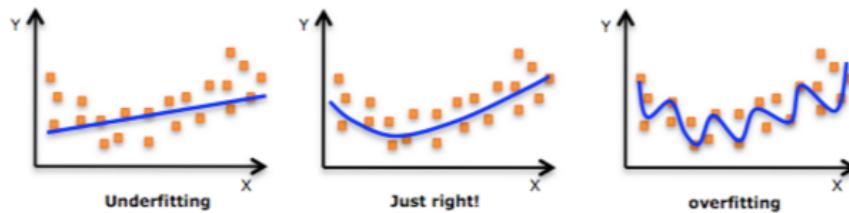


Figure 12.1: Overfitting y Underfitting

Es decir que el concepto de overfitting está asociado a la complejidad del modelo. Un modelo excesivamente complejo puede ajustar tan bien como queremos al set de entrenamiento pero funcionar muy bien para el set de test.

El concepto de underfitting es el opuesto y se produce cuando el modelo es demasiado simple. Si nuestros puntos están distribuidos en forma curva un

modelo lineal es demasiado simple, no tiene el poder expresivo necesario para representar correctamente el set de entrenamiento. Por lo tanto podemos decir que el modelo óptimo es aquel que tiene la complejidad necesaria para capturar lo que los datos expresan pero no mas.

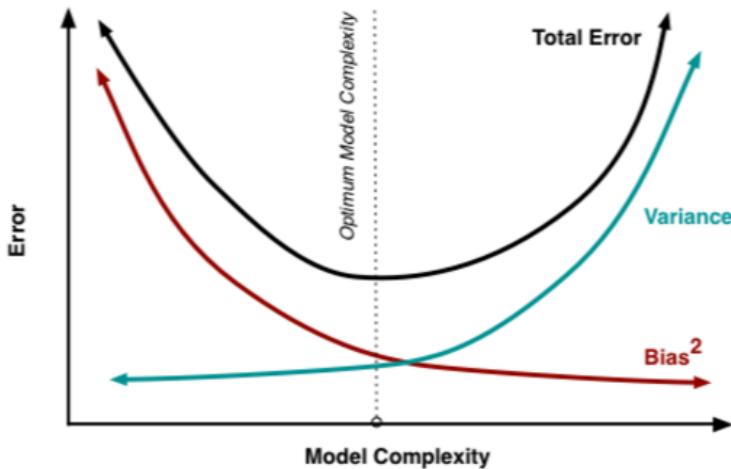


Figure 12.2: Bias y Variance

Los conceptos de *bias* y *variance* nos servirán para entender cuándo tenemos overfitting y cuándo underfitting. Podemos asociar el bias al error que tenemos en el set de entrenamiento y variance al error que tenemos en el set de test (o validación). A medida que aumenta la complejidad del modelo el error del set de entrenamiento disminuye, este es el bias que es siempre decreciente. Por otro lado a medida que aumentamos la complejidad el error del set de test aumenta. El óptimo está en algún punto medio. Cuando el modelo no es óptimo porque le falta complejidad tenemos alto bias, baja varianza y underfitting, cuando el modelo no es óptimo porque es demasiado complejo tenemos bajo bias, alta varianza y overfitting.

El bias puede asociarse al poder de *representación* del modelo y la varianza a la variabilidad del mismo. Cuando tenemos baja varianza y alto bias tenemos underfitting, el modelo no representa bien los datos pero tiene poca variabilidad. Cuando tenemos alta varianza y alto bias el modelo es un desastre, no entiende los datos y además tiene demasiada variabilidad. Si tenemos poco bias y poca varianza el modelo es el óptimo ajusta bien a los datos y generaliza bien. Finalmente si tenemos bajo bias y alta varianza tenemos un caso de overfitting.

En la práctica podemos identificar si tenemos un problema de bias o varianza ploteando el error del set de entrenamiento y del set de test en función de la cantidad de datos en el set de entrenamiento.

Cuando tenemos un problema de alto bias (underfitting) veremos que las curvas de error nunca se aproximan, esta distancia entre el error de test y de

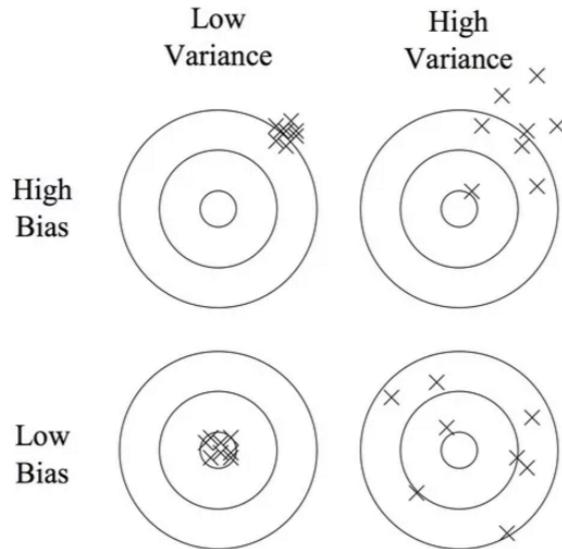


Figure 12.3: Bias y Variance

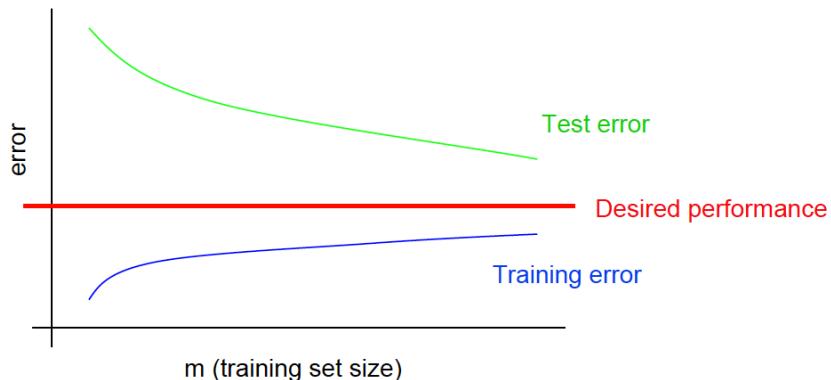


Figure 12.4: Alto Bias

entrenamiento es reflejo de un modelo que no tiene la complejidad necesaria para entender los datos.

Cuando el problema es la alta varianza tenemos que las curvas se aproximan pero el error total es demasiado alto, esto es un caso de overfitting, el modelo es demasiado complejo y generaliza mal.

En el caso de KNN el único hiper-parámetro que manejamos (además de

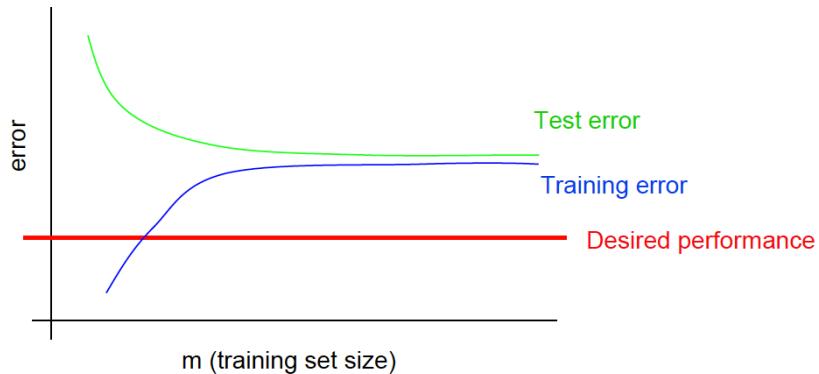


Figure 12.5: Alta Varianza

la métrica usada para la distancia) es k cuando k es un número muy grande KNN considera demasiados puntos para clasificar un punto nuevo, este es un caso de underfitting, nuestro algoritmo tiene *visión borrosa* por estar mirando demasiados puntos las fronteras entre nuestras clases se vuelven difusas y el poder expresivo del algoritmo es pobre. Por otro lado cuando k es muy bajo el algoritmo toma muy pocos puntos para clasificar, es un caso de overfitting, el modelo clasifica los puntos nuevos en base a muy poca evidencia, nuestro algoritmo alucina, las fronteras se vuelven muy complejas, demasiado complejas para el set de datos que tenemos. En KNN como el algoritmo no tiene entrenamiento no hace falta usar cross-validation y podemos probar diferentes valores de k directamente con el set de entrenamiento mismo curiosamente el mejor k no necesariamente es el que menor error nos da ya que sabemos que con $k = 1$ no tenemos errores. Queremos buscar el k mas grande con un error tan chico como los menores errores que vimos para todos los valores de k , este valor será en general el óptimo.

12.3 El Teorema de No Free Lunch

Este teorema publicado por [Wolpert] en el año 1997 es fuente de todo tipo de controversias y es posiblemente uno de los teoremas peor interpretados en el mundo de Machine Learning. En esta sección vamos a intentar explicar el significado del teorema y aclarar algunas interpretaciones erróneas que se hacen a partir del mismo.

Theorem 17 (No Free Lunch) *Dos algoritmos de optimización cualesquiera son equivalentes si los promediamos sobre el set de todos los problemas posibles.*

No vamos a demostrar el teorema aquí, para eso está el trabajo de Wolpert lo que si vamos a hacer es partiendo de que el teorema es cierto derivar un

corolario muy interesante:

Theorem 18 (No Free Lunch) *Dado un problema de optimización si un algoritmo funciona muy bien entonces existe un problema en el cual el algoritmo funciona igual de mal.*

Esto es fácil de demostrar, supongamos que, para un problema de clasificación binaria (1s y 0s), tenemos un set de entrenamiento y test, entrenamos a un algoritmo con el set de entrenamiento y obtenemos un 99% de precisión para el set de test. Si ahora cambiamos los labels del set de test invirtiendo 0s por 1s y 1s por 0s el mismo algoritmo tiene una efectividad del 1%.

Un corolario del corolario es:

Theorem 19 (No Free Lunch) *No existe un algoritmo que sea óptimo para cualquier problema de optimización.*

Hasta aquí vamos bien pero es posible que ya hayamos cometido un error fatal. Cuando nosotros pensamos en "cualquier problema posible" estamos pensando en problemas de ML que podemos querer resolver pero estos no son *todos* los problemas posibles. Todos los problemas posibles incluye el conjunto de problemas de ML que podemos querer resolver y el conjunto de todos los problemas que no tienen sentido y que nunca vamos a querer resolver. Visto de otra forma los problemas de ML que tienen sentido son solo un pequeño subespacio dentro del gigantesco espacio de todos los problemas posibles. Antes de terminar con un dolor de cabeza veamos un ejemplo sencillo:

Tenemos el siguiente set de entrenamiento en donde queremos predecir si nos va a gustar el capítulo 4 de una película basado en los 3 episodios anteriores:

$$\begin{aligned} (0,0,0) &\rightarrow 0 \\ (0,1,0) &\rightarrow 0 \\ (0,1,1) &\rightarrow 1 \\ (1,0,0) &\rightarrow 0 \\ (1,0,1) &\rightarrow 1 \\ (1,1,1) &\rightarrow 1 \end{aligned}$$

Es sencillo derivar un modelo predictor: si nos gustaron 2 o mas películas anteriores nos va a gustar la 4, si nos gustó solo una o ninguna entonces no.

Nuestro set de test es:

$$\begin{aligned} (0,0,1) &\rightarrow 0 \\ (1,1,0) &\rightarrow 1 \end{aligned}$$

Supongamos que nuestro algoritmo predice 0 y 1 es decir que tiene un 100% de efectividad. Si ahora cambiamos los labels del set de test:

$$\begin{aligned} (0,0,1) &\rightarrow 1 \\ (1,1,0) &\rightarrow 0 \end{aligned}$$

Nuestro algoritmo va a tener 0% de efectividad. Recordemos que el set de test no participa del entrenamiento por lo que el modelo será exactamente el mismo y las predicciones idénticas.

Nuestro primer problema es un problema lógico de ML, el segundo problema también es un problema de ML pero no tiene ningún sentido. Es como entrenar al modelo para aprender que si nos gustan dos o mas películas nos gustará la 4 y predecir exactamente lo contrario. Imaginemos en el caso de MNIST un algoritmo que aprenda a no-reconocer ningún dígito, es decir que dado un dígito de MNIST prediga cualquier número excepto el número representado. Esos problemas forman parte del espacio de todos los problemas posibles pero, claramente, no son problemas del mundo real. El teorema de Wolpert no aplica al mundo real sino al universo de todos los problemas, es importante desde el punto de vista teórico pero carece de funcionalidad en la práctica.

Hay una poderosa analogía entre el teorema NFL y el teorema fundamental de la compresión de datos. Ningún algoritmo de compresión puede comprimir cualquier archivo y ningún algoritmo de optimización puede optimizar cualquier problema. Sin embargo no queremos comprimir cualquier archivo ni optimizar cualquier problema. La mayoría de los archivos son random y no nos interesa comprimirlos. La mayoría de los problemas son random y no nos interesa optimizarlos!.

En definitiva es posible que UN algoritmo funcione mejor para cualquier problema de optimización que tenga sentido de la misma forma que hay UN algoritmo de compresión que suele ser el mejor en general (ej: PAQ). Es fundamental distinguir el concepto teórico del teorema NFL con el concepto fundamental, necesario y poderoso de que así como los archivos que comprimimos no son random los problemas de ML con los que trabajamos tampoco lo son.

Esta no es la primera ni la última vez que vamos a tener que distinguir entre el significado teórico de un teorema y su aplicación en la práctica, la clave es siempre la misma: **los datos no son random**

12.4 La Unión hace la fuerza: Ensambles

Los mejores algoritmos de ML suelen surgir de la combinación de varios algoritmos. Es muy raro que un solo algoritmo de ML logre mejores resultados que un ensamble. Esto puede verse en la mayoría de las competencias de ML en donde invariablemente los primeros resultados surgen de un ensamble. Esto no quiere decir que en la práctica siempre sea buena idea usar un ensamble, en algunos casos agregamos complejidad a cambio de una mejora muy pequeña en los resultados, en una competencia cualquier mejora es crítica pero en la vida real usar un algoritmo para tener una precisión del 80.55% o usar un ensamble de 200 algoritmos para una precisión de 80.71% probablemente sean resultados equivalentes y el segundo es muchísimo mas complejo.

En esta sección vamos a ver de forma abstracta los métodos teóricos para combinar algoritmos de ML que luego desarrollaremos en mayor detalle cuando veamos los algoritmos que los utilizan.

12.4.1 Bagging

Bagging en general implica aplicar el mismo clasificador n veces y luego promediar sus resultados para el resultado final. El problema es que aplicar el mismo clasificador n veces al mismo set de datos nos daría n veces el mismo resultado. Es por esto que *bagging* siempre se usa en conjunto con *bootstrapping*.

Bootstrapping consiste en tomar una muestra del set de entrenamiento del mismo tamaño del set de entrenamiento pero con reemplazo. Es decir que un dato puede estar varias veces en la muestra. Por ejemplo si tenemos los datos: Train = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 y tomamos muestras con reemplazo de tamaño 10 algunas muestras posibles son: $B1 = 4, 7, 2, 3, 4, 2, 7, 8, 7, 9$ o $B2 = 2, 6, 5, 5, 4, 9, 9, 7, 2, 6$. Podemos entonces entrenar nuestro clasificador con los bootstraps y obtener n clasificadores distintos. Luego podemos aplicar estos n clasificadores al set de datos original y promediar los resultados para obtener el resultado final, esto sirve tanto en problemas de regresión como de clasificación.

Como cada clasificador no ve el total de registros del set de entrenamiento la técnica de bagging disminuye notablemente las posibilidades de caer en overfitting, ya que ninguno de los n clasificadores individuales puede sobre-ajustar al set de entrenamiento completo.

Para cada clasificador existe un set de registros que queda afuera, por ejemplo en $B1$ los registros 0, 1, 5 y 6 no forman parte del bootstrap, a estos los llamamos registros *out of bag (OOB)*. La precisión de un clasificador que usa bagging se puede obtener mediante la clasificación de los registros OOB con el clasificador mismo, como si los registros OOB sirvieran como set de validación de cada clasificador individual.

Veamos el siguiente ejemplo: tenemos tres clasificadores y un set de entrenamiento de 10 registros para hacer clasificación binaria y marcamos en negrita los datos que forman parte del bootstrap para cada clasificador.

Reg	Class	c1	c2	c3	Ave
0	0	0	0	0	0
1	1	0	0	0	0
2	0	0	0	0	0
3	1	1	1	0	1
4	0	1	0	0	1
5	1	1	0	0	1
6	0	1	1	0	1
7	0	0	0	0	0
8	1	1	1	1	1
9	1	1	1	0	1
P		7/10	7/10	6/10	8/10
OOB		2/4	3/4	0/3	0.42

Table 12.1: Ejemplo Bagging.

El promedio de precisión para los registros OOB se puede usar para analizar la precisión del ensamble entero, de esta forma buscaremos los hiper-parámetros que nos den mejor promedio de precisión OOB, esto evita que la búsqueda de hiper-parámetros haga overfit.

Bagging puede usarse con cualquier algoritmo de clasificación o regresión pero es muy popular con árboles de decisión en donde da origen al algoritmo de *Random Forests* que veremos mas adelante.

12.4.2 Boosting

El método de boosting consiste en construir un algoritmo muy preciso a partir de un conjunto de algoritmos muy simples y que pueden funcionar bastante mal. El método consiste en entrenar un algoritmo simple, analizar sus resultados y luego entrenar otro algoritmo simple en donde se le da mayor peso a los resultados para los cuales el anterior tuvo peor performance. Cada algoritmo tiene a su vez un peso proporcional a la cantidad de aciertos que tuvo para el set de entrenamiento. El resultado final del algoritmo se construye mediante un promedio ponderado de todos los algoritmos usados con sus respectivos pesos.

El caso mas popular de boosting es el algoritmo Adaboost en donde se usar como algoritmo de clasificación a un simple separador del set de entrenamiento en 2, combinando y ponderando muchos de estos clasificadores se puede lograr un resultado bastante bueno.

Actualmente la técnica de boosting se ha vuelto mas importante en los algoritmos de tipo *gradient boosting* como GBM o XGBoost, que suelen estar en el top-5 entre los mejores algoritmos de clasificación y regresión.

12.4.3 Combinando Algoritmos Diferentes

En esta sección veremos formas de combinar el resultado de diferentes algoritmos de ML para obtener un resultado final, estos procesos suelen ser la clave para obtener un mejor resultado ya que permiten aprovechar el poder expresivo de varios modelos completamente diferentes para obtener un resultado común. Para evitar confusiones vamos a concentrarnos en problemas de clasificación, con la aclaración.

Majority Voting

Empecemos por el caso mas simple: Tenemos varios clasificadores distintos para un cierto problema, cada uno de ellos produce un resultado y queremos obtener un resultado final. Una aproximación simple es ver cual es la clase que tiene mayoría entre todos los clasificadores. Este tipo de ensamble tiene sentido cuando la predicción es directamente la clase, si la predicción es la probabilidad de cada clase entonces otros métodos funcionan mejor. Hasta aquí todo parece muy simple pero hay algunos detalles a considerar.

El resultado del voto por mayoría mejora notablemente si se usan resultados que tengan poca correlación entre sí. Veamos un ejemplo, suponemos que todos

los datos tienen clase real 1 y mostramos el resultado de 3 clasificadores para nuestro set de test.

```
1111111100 = 80\%
1111111100 = 80\%
1011111100 = 70\%
-----
1111111100 = 80\%
```

Usando tres clasificadores cuyas predicciones están muy correlacionadas es difícil obtener una nueva predicción mejor que las anteriores. Veamos ahora qué pasa cuando los modelos no están tan de acuerdo entre sí.

```
1111111100 = 80\%
0111011101 = 70\%
1000101111 = 60\%
-----
1111111101 = 90\%
```

Cuando la correlación entre los modelos es baja el resultado del ensamble en general mejora notablemente el resultado de cada modelo individual. Una primera conclusión es que dados muchos clasificadores es conveniente elegir un conjunto que tenga buenos resultados y que estén muy poco correlacionados. Este será luego el set de clasificadores que usaremos para el ensamble.

Los modelos del ensamble pueden tener diferentes niveles de precisión, es lógico pensar que el modelo que mejor está funcionando es el que en general tiene mayor cantidad de aciertos y por lo tanto solo queremos cambiar sus predicciones si muchos de los demás clasificadores mas débiles opinan lo contrario. Esto es simplemente darle un *peso* a cada modelo. Un esquema simple es darle n votos al mejor clasificador, $n - 1$ votos al siguiente, etc. Luego simplemente aplicamos majority voting pero el mejor clasificador aparece varias veces en la votación y por lo tanto su opinión será mas importante.

Averaging

Promediar el resultado de varios clasificadores es un método muy popular que funciona en muchos problemas distintos: regresión, clasificación ya sea para predecir clases o probabilidades, etc. La idea principal es reducir el overfitting. En general una separación suave entre las clases es mejor que una separación muy irregular y promediar clasificadores logra esto.

La línea negra es una separación mejor que la verde, promediando muchas separaciones verdes deberíamos lograr una separación similar a la línea negra.

Cuando promediamos clasificadores que predicen la probabilidad de las clases hay que tener cuidado porque cada clasificador individual puede tener una calibración completamente diferente, uno puede dar probabilidades muy cercanas a 1s y 0s mientras que otro a lo mejor se mantiene dentro de un cierto rango. Una solución para esto es convertir cada probabilidad de un rango entre 1 y n siendo

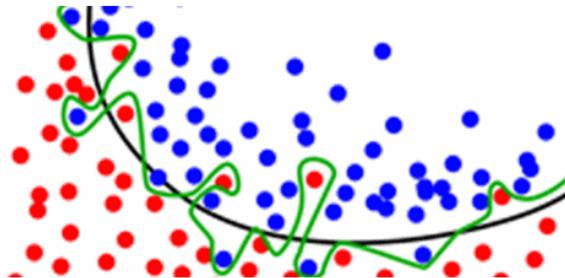


Figure 12.6: Averaging

n el total de registros. El registro con mayor probabilidad tiene 1, el segundo 2 y el de menor probabilidad n sin importar el valor de las probabilidades. Si hacemos esto para todos los clasificadores podemos luego promediar los rangos y convertir estos promedios en un número entre 0 y 1 para la probabilidad final.

Ejemplo:

Reg	c1	c1 rank	c2	c2 rank	Ave	Final Prob
1	0.57	3	0.3605	1	2	0.25
2	0.04	4	0.3502	3	3.5	1
3	0.96	2	0.35	4	3	0.75
4	0.99	1	0.36	2	1.5	0

Table 12.2: Promedio por Rango.

Para calcular la probabilidad final simplemente normalizamos $prob = \frac{x - min}{max - min}$.

Blending

El proceso de *blending* es uno de los que mejores resultados da en la creación de ensambles a partir de clasificadores diferentes. La idea es entrenar varios clasificadores diferentes y armar un set de datos con sus predicciones para luego entrenar otro clasificador que realice las predicciones finales en base a la combinación de los clasificadores. Esto puede sonar un poco confuso así que vamos a entrar en algunos detalles.

Para realizar el blending vamos a separar una pequeña parte del set de entrenamiento (10%), lo que resta lo vamos a usar para entrenar a los n clasificadores diferentes, notar que el set separado no debe usarse para validación, para validar hay que hacer un split de lo que quedó del set de entrenamiento como hacíamos habitualmente. Una vez entrenados los clasificadores los vamos a usar para predecir sobre el set que separamos (10%), el resultado es un nuevo dataset en donde tenemos un 10% de los registros y n predicciones para los mismos.

Para estos registros conocemos cuál es su verdadera clase porque formaban parte del set de entrenamiento original. Lo que vamos a hacer entonces es entrenar un nuevo modelo sobre estas predicciones para que el modelo aprenda como combinar los resultados de los clasificadores de forma tal que estos nos den la predicción final. Este modelo en general es simple, puede usarse logistic regression o gbm.

El proceso de blending funciona mejor cuanto mas grande es el set de entrenamiento original porque permite que el set usado para entrenar el blender sea mayor, en estos casos podríamos usar un modelo aún mas complejo para realizar el blend, es decir un modelo no-lineal.

Una vez que entrenamos el blender lo que hacemos es re-entrenar los clasificadores con el set de entrenamiento completo (mas datos siempre dan mejores resultados) y luego aplicar esos modelos al set de test para finalmente aplicar el blender para obtener las predicciones finales.

Algo que puede hacerse y que suele traer buenos resultados es una vez que tenemos las predicciones para el set de test combinar los sets de entrenamiento y test en un nuevo set de entrenamiento y repetir el proceso completo con este nuevo súper-set de entrenamiento. Es decir separamos un 10% de los registros, entrenamos los modelos, entrenamos al blender en base a las predicciones del 10% que separamos, re-entrenamos los modelos con el súper-set completo, aplicamos los modelos para predecir el set de test y luego aplicamos el blender para las predicciones finales. La conclusión final es que todo esto toma tiempo!

Blending

1. Separar un 10% del set de entrenamiento
2. Entrenar n clasificadores diferentes con el 90% del set de entrenamiento (separando este 90% en sets de entrenamiento y validación para optimizar cada modelo)
3. Realizar n predicciones para el 10% que separamos usando cada uno de los n clasificadores que entrenamos.
4. Entrenar un modelo *blender* que use las predicciones aprendidas para realizar la clasificación final
5. Entrenar los n modelos con el set de entrenamiento completo
6. Realizar predicciones con estos modelos para el set de test
7. Combinar las predicciones para el set de test usando el modelo blender
8. Combinar los sets de entrenamiento y test en un nuevo súper-set de entrenamiento en donde los labels para lo que era el set de test son los que predijo el blender.
9. Separar un 10% del súper-set

10. Entrenar los n modelos en el 90% restante del súper-set
11. Predecir los resultados para el 10% que separamos usando cada modelo
12. Entrenar un blender para combinar estas predicciones en la predicción final
13. Entrenar los n modelos usando el súper-set completo
14. Aplicar los modelos aprendidos al set de test
15. Aplicar el súper-blender al set de test

12.5 Teoría de Machine Learning

En esta sección vamos a formalizar un poco la teoría de Machine Learning. Los conceptos que veremos fueron enunciados por primera vez en 1984 por Leslie Valiant [Valiant].

El objetivo es presentar algunas definiciones y teoremas que son útiles para comprender conceptos que luego son fundamentales en la práctica como *bias* y *variance*. Curiosamente esta sección podría estudiarse antes o después del resto de los temas del capítulo. Aquellos que lean esta sección primero tendrán una base teórica que les va a permitir entender los conceptos que explicaremos mas adelante sin necesidad de "creer" en ellos. Los que lean la sección al final tendrán una explicación de cosas que fueron aprendiendo, es decir una forma de develar varios misterios. Dejamos la decisión a consideración de cada lector. Por si están pensando en decidir en base al lanzamiento de una moneda vamos a empezar con un concepto de probabilidades que vamos a necesitar luego.

12.5.1 Desigualdad de Hoeffding

Sean Z_1, \dots, Z_m m variables aleatorias independientes que responden, todas ellas, a una distribución tipo Bernoulli(ϕ). Esto quiere decir que $P(Z_i = 1) = \phi$ y $P(Z_i = 0) = 1 - \phi$. Tomemos el promedio de todas estas variables como $\hat{\phi} = \frac{1}{m} \sum_{i=1}^m Z_i$ y sea $\gamma > 0$ una constante. Entonces:

$$P(|\phi - \hat{\phi}| > \gamma) \leq \exp(-2\gamma^2 m) \quad (12.1)$$

Esta definición a la cuál se la llama también *Chernoff bound* dice que si tomamos el promedio de m variables de tipo Bernoulli, entonces la probabilidad de que el promedio sea diferente del valor real ϕ es pequeña para un m grande. Esto es simplemente una conclusión de la ley de los grandes números, si tenemos una moneda cargada, podemos estimar su ϕ lanzándola una cantidad m de veces grande y calculando el promedio de veces que salió 1.

Esta desigualdad es parte fundamental de la teoría de Machine Learning cosa que, por supuesto, veremos mas adelante.

12.5.2 Error de Entrenamiento

Vamos a restringir el análisis a problemas de clasificación binarios, es decir con dos clases $\{0, 1\}$. Todo lo que vamos a desarrollar se puede extender sin dificultad a problemas de regresión o de clasificación multiclase.

Nuestro set de datos tendrá entonces m filas x_i de n dimensiones y m labels $y_i \in \{0, 1\}$. Llamaremos al set de entrenamiento $S = \{(x_1, y_1), \dots, (x_m, y_m)\}$ y suponemos que los datos del set de entrenamiento surgen de una cierta distribución de probabilidad \mathcal{D} .

Al resultado del algoritmo que entrenemos lo vamos a llamar nuestra *hipótesis* (h). Vamos a definir el *error de entrenamiento*, *error empírico* o *riesgo empírico* de nuestra hipótesis h de la siguiente forma:

$$\hat{\epsilon}(h) = \frac{1}{m} \sum_{i=1}^m \mathbb{1}\{h(x_i) \neq y_i\} \quad (12.2)$$

Esto no es ni mas ni menos que la fracción de registros que clasificamos mal, es decir que si tenemos 100 datos y clasificamos mal a 12 de ellos entonces el error de entrenamiento es 0.12.

12.5.3 Error de Generalización

Un algoritmo de clasificación es evaluado de acuerdo a que tan bien puede clasificar a datos que no pertenecen al set de entrenamiento, es decir datos nuevos. Para que el algoritmo pueda hacer esto bien es necesario que el algoritmo aprenda no solo a ajustar al set de entrenamiento sino a *generalizar*. Si aplicamos el algoritmo a datos que no pertenecen al set de entrenamiento tenemos lo que llamamos *Error de generalización*

$$\epsilon(h) = P_{(x,y) \sim \mathcal{D}}(h(x) \neq y) \quad (12.3)$$

Es decir que si tomamos un dato cualquiera de nuestra distribución \mathcal{D} y medimos la probabilidad de clasificarlo mal.

12.5.4 Overfitting y Underfitting

Vamos a definir como *underfitting* al caso en el cual nuestra hipótesis tiene un error de entrenamiento $\hat{\epsilon}$ alto.

Vamos a definir como *overfitting* al caso en el cual nuestra hipótesis tiene un error de entrenamiento $\hat{\epsilon}$ bajo pero un error de generalización ϵ alto.

12.5.5 El Problema de Clasificación

El objetivo de un algoritmo es encontrar los parámetros θ asociados a la mejor hipótesis h ; habiendo definido el error de entrenamiento y el error de generalización una forma de encontrar estos parámetros es minimizando el error de entrenamiento:

$$\hat{\theta} = \arg \min_{\theta} \hat{e}(h_{\theta}) \quad (12.4)$$

A esto lo llamamos *minimización empírica del riesgo* (ERM) por sus siglas en inglés. La hipótesis que resulta es $h = h_{\theta}$. ERM es la forma más básica de definir un algoritmo de entrenamiento y es la que vamos a usar en el resto de la sección. Todos los algoritmos pueden aproximarse a un problema ERM por lo que no perdemos generalidad.

12.5.6 El Espacio de Hipótesis

Para independizarnos de los parámetros asociados a una hipótesis vamos a definir el concepto de la *familia de hipótesis* \mathcal{H} como el conjunto de todos los clasificadores posibles que puede generar un algoritmo. Por ejemplo si nuestro algoritmo es un clasificador lineal entonces $\mathcal{H} = \{h_{\theta} : h_{\theta}(x) = 1\{\theta * x \geq 0\}, \theta \in \mathbb{R}^{n+1}\}$. Es decir el conjunto de todos los clasificadores en donde la frontera es lineal. La cantidad de parámetros es $n + 1$ porque agregamos el término independiente. Si nuestro algoritmo fuera otro, por ejemplo un árbol de decisión entonces \mathcal{H} sería el conjunto de todos los árboles de decisión posibles.

12.5.7 Minimización Empírica del Riesgo en \mathcal{H}

Podemos definir ahora al entrenamiento como la hipótesis que minimiza el riesgo empírico.

$$\hat{h} = \arg \min_{h \in \mathcal{H}} \hat{e}(h) \quad (12.5)$$

12.5.8 Cuando $|\mathcal{H}|$ es finito

Consideremos primero el caso en el cual la cantidad de hipótesis posibles es un número finito. Es decir que tenemos $\mathcal{H} = \{h_1, \dots, h_k\}$ para un total de k hipótesis posibles. En este caso \mathcal{H} es un conjunto de k funciones que mapean X a $\{0, 1\}$. La minimización empírica del riesgo implica seleccionar \hat{h} como la función dentro de las k funciones posibles que minimiza el error de entrenamiento.

Queremos derivar cual es la relación que existe entre el error de entrenamiento \hat{h} y el error de generalización h .

Consideremos una hipótesis cualquiera fija $h_i \in \mathcal{H}$. Tomemos un dato cualquiera que no pertenezca al set de entrenamiento de nuestra distribución $(x, y) \mathcal{D}$. Definimos ahora $Z = 1\{h_i(x) \neq y\}$ es decir que Z vale 1 si clasificamos mal el dato y vale 0 si lo clasificamos bien. Este Z es el error de generalización y puede tomar valores 0 o 1 por lo que podemos decir que es una distribución de tipo Bernoulli.

Por otro lado podemos aplicar la hipótesis h_i a cualquier punto $x_i \in S$ de nuestro set de entrenamiento. Y definir $Z_j = 1\{h_i(x_j) \neq y_j\}$. Esta es otra distribución Bernoulli.

Como tanto los puntos del set de entrenamiento como los puntos del set de validación o test fueron tomados de la misma distribución \mathcal{D} entonces Z y Z_j tendrán la misma distribución.

Podemos escribir el error de entrenamiento como el valor esperado de Z_j :

$$\hat{\epsilon}(h_i) = \frac{1}{m} \sum_{j=1}^m Z_j \quad (12.6)$$

Podemos aplicar ahora la desigualdad de Hoeffding (12.1):

$$P(|\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma) \leq 2 \exp(-2\gamma^2 m) \quad (12.7)$$

Esto nos dice que para una hipótesis en particular h_i el error de entrenamiento va a ser cercano al error de generalización con una alta probabilidad cuando m es grande. Queremos ahora probar que esto es válido para cualquier hipótesis posible. Sabemos que la probabilidad de la unión de n variables aleatorias es menor o igual a la suma de sus probabilidades es decir: $P(A_1 \cup \dots \cup A_n) \leq P(A_1) + \dots + P(A_n)$.

Definamos a $P(A_i) \leq 2 \exp(-2\gamma^2 m)$

$$\begin{aligned} P(\exists h \in \mathcal{H} / |\epsilon(h) - \hat{\epsilon}(h)| > \gamma) &= P(A_1 \cup \dots \cup A_k) \\ &\leq \sum_{i=1}^k P(A_i) \\ &\leq \sum_{i=1}^k 2 \exp(-2\gamma^2 m) \\ &= 2k \exp(-2\gamma^2 m) \end{aligned} \quad (12.8)$$

Si restamos de 1 a ambos lados tenemos:

$$\begin{aligned} P(\nexists h \in \mathcal{H} / |\epsilon(h) - \hat{\epsilon}(h)| > \gamma) &= P(\forall h \in \mathcal{H} / |\epsilon(h) - \hat{\epsilon}(h)| \leq \gamma) \\ &\geq 1 - 2k \exp(-2\gamma^2 m) \end{aligned} \quad (12.9)$$

Por lo tanto con probabilidad igual al menos a $1 - 2k \exp(-2\gamma^2 m)$ tenemos que $\epsilon(h)$ va a estar a lo sumo a γ de $\hat{\epsilon}(h)$. Y esto vale para cualquier $h \in \mathcal{H}$. Esto se llama *convergencia uniforme* ya que vale simultáneamente para todas las hipótesis posibles.

En el desarrollo anterior usamos tres constantes: γ , m y la probabilidad del error. Podemos expresar cualquiera de ellas en función de las otras dos.

Por ejemplo supongamos que dado un cierto γ y $\delta > 0$ queremos saber que tan grande tiene que ser el set de entrenamiento m de forma tal que podamos garantizar que con probabilidad $1 - \delta$ el error de entrenamiento va a estar a lo sumo a γ del error de generalización.

Aplicamos entonces $\delta = 2l \exp(-2\gamma^2 m)$ y despejamos m para encontrar:

$$m \geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta} \quad (12.10)$$

Por lo tanto con probabilidad al menos $1 - \delta$ tenemos que $|\epsilon(h) - \hat{\epsilon}(h)| \leq \gamma$ para cualquier $h \in \mathcal{H}$. Esto es lo mismo que decir que la probabilidad de que $|\epsilon(h) - \hat{\epsilon}(h)| \geq \gamma$ para algún $h \in \mathcal{H}$ es a lo sumo δ .

Esta cota nos da una idea de cuantos datos necesitamos en el set de entrenamiento. Al valor de m que necesitamos para que un algoritmo alcance una cierta cota de error con respecto al error de generalización lo llamamos *complejidad de la muestra* (sample complexity).

La clave de la cota que hemos calculado es que la cantidad de datos en el set de entrenamiento crece de forma **logarítmica** con respecto a k es decir a la cantidad de hipótesis.

De la misma forma podemos partir de valores de m y δ fijos y resolver γ esto nos sirve para mostrar que con probabilidad $1 - \delta$ para cualquier hipótesis $h \in \mathcal{H}$:

$$|\hat{\epsilon}(h) - \epsilon(h)| \leq \sqrt{\frac{1}{2m} \log \frac{2k}{\delta}} \quad (12.11)$$

Suponiendo que la convergencia uniforme es cierta es decir que $|\epsilon(h) - \hat{\epsilon}(h)| \leq \gamma$ para todo $h \in \mathcal{H}$. ¿Qué podemos saber sobre el error de generalización del algoritmo que elige $\hat{h} = \arg \min_{h \in \mathcal{H}} \hat{\epsilon}(h)$

Definimos a nuestra mejor hipótesis como h^* es decir $h^* = \arg \min_{h \in \mathcal{H}} \epsilon(h)$. Notemos que estamos definiendo a nuestra mejor hipótesis como aquella que minimiza el error de generalización.

$$\epsilon(\hat{h}) \leq \hat{\epsilon}(\hat{h}) + \gamma \quad (12.12)$$

Esto lo sabemos porque $|\epsilon(\hat{h}) - \hat{\epsilon}(\hat{h})| \leq \gamma$ por la convergencia uniforme.

Luego:

$$\epsilon(\hat{h}) \leq \hat{\epsilon}(h^*) + \gamma \quad (12.13)$$

Porque h minimiza $\hat{\epsilon}(h)$ y $\hat{\epsilon}(\hat{h}) \leq \hat{\epsilon}(h)$ para todo h , por lo tanto en particular $\hat{\epsilon}(h) \leq \hat{\epsilon}(h^*)$. Es decir que el error de entrenamiento de h^* no puede ser mejor que el de \hat{h} que es la hipótesis que minimiza el error de entrenamiento.

Finalmente:

$$\epsilon(\hat{h}) \leq \epsilon(h^*) + 2\gamma \quad (12.14)$$

Aquí usamos nuevamente la convergencia uniforme para mostrar que $\hat{\epsilon}(h^*) \leq \epsilon(h^*) + \gamma$.

Lo que hemos demostrado es que si se da la convergencia uniforme entonces el error de generalización de \hat{h} es a lo sumo 2γ peor que la mejor hipótesis posible de \mathcal{H} (!)

Este es quizás el resultado más importante que hemos obtenido hasta ahora y es de tremenda significancia porque lisa y llanamente decimos que si tenemos un set de entrenamiento lo suficientemente grande entonces el resultado de un cierto algoritmo sobre ese set de entrenamiento está muy cerca del mejor resultado posible que el algoritmo nos pueda dar. En otras palabras el entrenamiento del algoritmo justifica que podamos usarlo para generalizar a cualquier otro dato (!)

Es hora de poner todo esto en un teorema:

Theorem 20 (Generalización) *Sea $|\mathcal{H}| = k$ y sean m, δ constantes. Entonces con probabilidad al menos $1 - \delta$*

$$\epsilon(\hat{h}) \leq (\min_{h \in \mathcal{H}} \epsilon(h)) + 2\sqrt{\frac{1}{2m} \log \frac{2k}{\delta}} \quad (12.15)$$

Esto se demuestra facilmente haciendo que γ sea igual al término de la raíz cuadrada, usando la presunción de que la convergencia uniforme ocurre con probabilidad al menos $1 - \delta$ y teniendo en cuenta que la convergencia uniforme implica que $\epsilon(h)$ es a lo sumo 2γ mayor que $\epsilon(h^*) = \min_{h \in \mathcal{H}} \epsilon(h)$ (lo cual ya demostramos).

El teorema de la generalización cuantifica el concepto de bias y variance. Por ejemplo supongamos que tenemos un algoritmo que tiene una familia de hipótesis \mathcal{H} y que consideramos cambiar a otro algoritmo cuya familia de hipótesis es \mathcal{H}' con $\mathcal{H} \subseteq \mathcal{H}'$. Es decir que nuestro nuevo algoritmo tiene mayor cantidad de hipótesis posibles, es decir que es un algoritmo mas complejo.

El primer término de 12.15 ($\min_h \epsilon(h)$) solo puede disminuir porque estamos tomando el mínimo sobre una cantidad de funciones mayor. Por lo tanto al aprender de un algoritmo mas complejo el *bias* solo puede disminuir. Sin embargo si k aumenta el segundo término ($2\sqrt{\cdot}$) también aumenta. Este incremento es lo que llamamos *varianza* aumentando cuando nuestro algoritmo es mas complejo.

Si mantenemos γ y δ constantes y despejamos m como hicimos antes podemos obtener la siguiente cota a la complejidad muestral:

Theorem 21 (Corolario al teorema de generalización) *Sea $|\mathcal{H}| = k$ y sean δ, γ constantes. Entonces para que valga $\epsilon(\hat{h}) \leq \min_{h \in \mathcal{H}} \epsilon(h) + 2\gamma$ con probabilidad al menos $1 - \delta$ es condición suficiente que:*

$$\begin{aligned}
m &\geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta} \\
&= O\left(\frac{1}{\gamma^2} \log \frac{k}{\delta}\right)
\end{aligned} \tag{12.16}$$

12.5.9 Cuando $|\mathcal{H}|$ es infinito

Tenemos que considerar el caso de algoritmos que generan un conjunto de hipótesis infinitas. Un algoritmo de clasificación lineal, por ejemplo, puede generar infinitas rectas para separar nuestras clases. Queremos ver si lo que hemos demostrado hasta ahora puede extenderse a este caso.

Empecemos con una forma bastante incorrecta de encarar el problema. Vamos a argumentar que en una computadora no puede existir un algoritmo que genere una cantidad infinita de hipótesis.

Sea \mathcal{H} una familia de hipótesis parametrizada por un conjunto de d números reales. Dado que tenemos que usar una computadora para representar estos números podemos decir que cada uno de ellos es un flotante de 64 bits. Por lo tanto necesitamos $64d$ bits por cada hipótesis. Esto quiere decir que el total de hipótesis posibles es $k = 2^{64d}$. Por lo tanto podemos aplicar todo lo que vimos en el caso anterior.

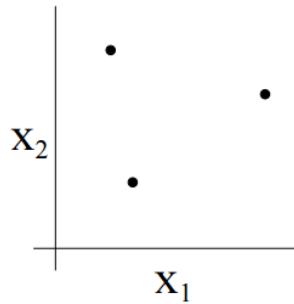
Aplicando el corolario del teorema de generalización (12.16) si queremos garantizar que $\epsilon(\hat{h}) \leq \epsilon(h^*) + 2\gamma$ con probabilidad $1 - \delta$ entonces es suficiente con garantizar que $m \geq O\left(\frac{1}{\gamma^2} \log \frac{2^{64d}}{\delta}\right) = O\left(\frac{d}{\gamma^2} \log \frac{1}{\delta}\right) = O(d)$. Por lo tanto la cantidad de datos en el set de entrenamiento que necesitamos crece de forma lineal a la cantidad de parámetros del modelo. Este es otro resultado sumamente importante.

Uno de los detalles que no hemos considerado hasta el momento es el de la parametrización de \mathcal{H} hemos asumido que un clasificador lineal necesita $n+1$ parámetros, es decir una dimensión mas que el set de entrenamiento debido al término independiente. Esto asume que la hipótesis es de tipo $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$ pero esa no es la única parametrización posible de una recta, podríamos escribir por ejemplo $h_{u,v}(x) = (u_0^2 - v_0^2) + (u_1^2 - v_1^2)x_1 + \dots + (u_n^2 - v_n^2)x_n$ que también es una recta pero en lugar de $n+1$ parámetros tiene $2n+2$. Vamos entonces a usar un concepto llamado la dimensión VC (Vapnik-Chervonenkis) para formalizar la parametrización de un algoritmo de clasificación.

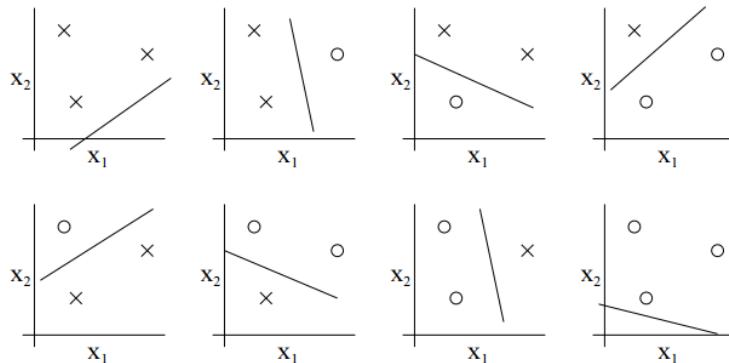
Dada una familia de hipótesis \mathcal{H} definimos la dimensión de Vapnik-Chervonenkis de \mathcal{H} denotada $VC(\mathcal{H})$ como la cardinalidad máxima de un set de puntos que la hipótesis pueda clasificar para cualquier repartición de los labels. Si el algoritmo puede separar cualquier cantidad de puntos entonces $VC(\mathcal{H}) = \infty$

Consideremos un clasificador lineal y el siguiente conjunto de tres puntos en el plano:

Podemos probar que sea cual fuera la distribución de los labels de esos tres puntos siempre existe una recta que separa las dos clases. El siguiente diagrama

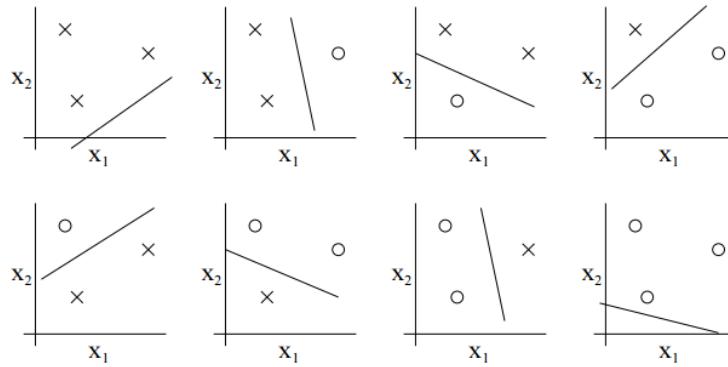
Figure 12.7: Dimensión VC para un clasificador lineal en \mathbb{R}^2

muestra todos los casos posibles y la separación.

Figure 12.8: Dimensión VC para un clasificador lineal en \mathbb{R}^2

Notemos que algunos conjuntos de 3 puntos en el plano no pueden separarse con una recta, por ejemplo los de la figura 12.9, pero esto no impide que $VC(\mathcal{H})$ sea 3, alcanza con encontrar **una** configuración de m puntos en n dimensiones que podamos separar independientemente de como rotulemos a cada punto para que $VC(\mathcal{H}) = m$

Es posible mostrar que no existe **ninguna** configuración de cuatro puntos en el plano de forma tal que una recta pueda separarlos independientemente de como rotulemos a cada punto. Por ejemplo si ponemos los cuatro puntos en los vértices de un cuadrado rotulando con "x" y "o" las esquinas opuestas tenemos el famoso problema del XOR que es imposible de separar mediante un plano. Por lo tanto en nuestro caso para la familia de hipótesis de los clasificadores lineales en \mathbb{R}^2 tenemos $VC(\mathcal{H}) = 3$. Repasando llegamos a 3 porque existe una configuración de 3 puntos tales que siempre son separables independientemente

Figure 12.9: Dimensión VC para un clasificador lineal en \mathbb{R}^2

de como estén rotulados y no existe una configuración de 4 puntos que cumpla lo mismo.

Esto nos lleva a sacar algunas conclusiones interesantes sobre la dimensionaldad y la capacidad de un algoritmo de clasificar:

Theorem 22 (Radon) *Cualquier conjunto de $d + 2$ puntos en \mathbb{R}^d puede ser particionado en dos conjuntos disjuntos de forma tal que sus fronteras convexas se intersecten. Si las fronteras convexas se intersectan entonces los puntos no se pueden separar linealmente.*

Este teorema podríamos haberlo usado para demostrar que en \mathbb{R}^2 no podemos separar linealmente 4 puntos. Por el teorema de Radon la dimensión VC de un conjunto de puntos en \mathbb{R}^d es $d + 1$.

Otro teorema útil relacionado es el siguiente:

Theorem 23 (Cover) *Dado un conjunto de m puntos en d dimensiones. Existe, con una alta probabilidad, una transformación que proyecta a los puntos en $k \gg d$ dimensiones de forma tal que los puntos sean linealmente separables.*

Este teorema se puede demostrar fácilmente, si tenemos m puntos podemos proyectarlos al simplex de $m - 1$ dimensiones de forma tal que cada punto ocupe una esquina del mismo. Por ejemplo 4 puntos en las esquinas de un tetraedro. Si hacemos esto para cualquier asignación de dos labels a estos puntos tenemos un hiperplano separador. Es decir que si asignamos 1 o 0 a cada una de las 4 esquinas de un tetraedro siempre podemos encontrar un hiperplano que separe los 1s de los 0s.

El teorema de Cover es la justificación a todos los métodos basados en el uso de Kernels, en concreto cuando proyectamos a los puntos a un espacio de muchas dimensiones es muy probable que podamos encontrar una separación lineal para los mismos.

Habiendo estudiado la dimensión VC podemos enunciar ahora el siguiente teorema:

Theorem 24 (Vapnik) *Sea \mathcal{H} una familia de hipótesis con $VC(\mathcal{H}) = d$. Entonces con probabilidad al menos $1 - \delta$ tenemos que para todo $h \in \mathcal{H}$:*

$$|\epsilon(h) - \hat{\epsilon}(h)| \leq O\left(\sqrt{\frac{d}{m} \log \frac{m}{d}} + \frac{1}{m} \log \frac{1}{\delta}\right) \quad (12.17)$$

Y con probabilidad al menos $1 - \delta$ sabemos que:

$$\hat{\epsilon}(h) \leq \epsilon(h^*) + O\left(\sqrt{\frac{d}{m} \log \frac{m}{d}} + \frac{1}{m} \log \frac{1}{\delta}\right) \quad (12.18)$$

Esto quiere decir que si una hipótesis tiene una dimensión VC finita entonces la convergencia uniforme ocurre cuando m se vuelve grande. Como antes esto nos permite establecer una cota para el error de generalización.

Theorem 25 (Corolario al teorema de Vapnik) *Para que valga $|\epsilon(h) - \hat{\epsilon}(h)| \leq \gamma$ para todo $h \in \mathcal{H}$ con probabilidad al menos $1 - \delta$ es suficiente que $m = O(d)$*

Es decir que la cantidad de datos en el set de entrenamiento para aprender "bien" usando \mathcal{H} es lineal en la dimensión VC de \mathcal{H} . En general y asumiendo una parametrización razonable esto es también lineal a la cantidad de parámetros. Poniendo todo en contexto podemos concluir que para que un algoritmo funcione bien necesitamos tener al menos una cantidad de datos que sea lineal de acuerdo a la cantidad de parámetros del mismo.

El teorema de Vapnik junto con su corolario es la pieza mas importante de toda la teoría de Machine Learning, ya que nos demuestra que un algoritmo puede aprender a partir del set de entrenamiento y generalizar con un error acotado al error de entrenamiento siempre y cuando la cantidad de datos del set de entrenamiento sea lineal a la dimensión VC del algoritmo.

Beautiful.

12.5.10 El Principio de la Mínima Descripción

El principio de la mínima descripción (MDL) por sus siglas en inglés [Rissanen] [Grumwald] es una forma de interpretar los algoritmos de Machine Learning desde el punto de vista de la compresión de datos. La idea es muy simple: cuanto mas se puede comprimir un set de datos mas se puede aprender del mismo.

De acuerdo a MDL hay dos cosas para representar: los datos y el modelo. El modelo ideal, es decir el mejor algoritmo, es aquel que minimiza la sumatoria de la longitud necesaria en bits para representar los datos y la longitud para representar el modelo.

$$MDL = \arg \min L(X) + L(\theta) \quad (12.19)$$

En MDL buscamos un algoritmo que tenga un bajo error de entrenamiento y que al mismo tiempo sea simple. Por cada dato de nuestro set de datos un algoritmo de ML realiza una cierta predicción, en base a la predicción podemos calcular la probabilidad del dato y en base a la probabilidad del dato podemos calcular la longitud ideal en bits del mismo como $\log_2(P)$.

Consideremos por ejemplo un simple problema de regresión en donde tenemos un conjunto de puntos (x, y) y queremos encontrar un algoritmo que dado x nos permita aproximar y . Un algoritmo de regresión va a recibir un punto x y va a devolver una estimación para el valor de y , podemos convertir esta estimación en una probabilidad si suponemos que los residuos se distribuyen de forma normal:

$$P(\epsilon_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\epsilon_i^2}{2\sigma^2}\right) \quad (12.20)$$

Esto nos lleva a:

$$P(y_i|x_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - \theta x_i)^2}{2\sigma^2}\right) \quad (12.21)$$

Notemos que minimizar el error es equivalente a maximizar la productoria de todas estas probabilidades (likelihood) o lo que es lo mismo la sumatoria del logaritmo de estas probabilidades (log-likelihood)

A modo de ejemplo consideremos que probamos para un cierto problema de regresión con un modelo lineal y modelos polinómicos de grado n . A medida que aumentamos el grado de los polinomios vamos a ajustar mejor a los puntos pero tal vez el modelo sea demasiado complejo y estemos cayendo en un caso de overfitting.

La figura 12.10 muestra el resultado de ajustar una recta, un polinomio de grado 2 y un polinomio de grado 6 a un cierto conjunto de puntos. Podemos calcular la complejidad de cada modelo calculando la sumatoria de la probabilidad para la predicción que cada modelo realiza sobre los puntos y agregando la complejidad del modelo como muestra la siguiente tabla:

p1 probs	p1 bits	p2 probs	p2 bits	p6 probs	p6 bits
0.0796	3.65	0.9975	0.0037	0.999	0.001
0.9382	0.0920	0.9382	0.092	0.9677	0.02
...

Table 12.3: Cantidad de bits para tres modelos diferentes

Como podemos ver a medida que el grado del polinomio es mas alto mejor ajustamos a los puntos y por lo tanto tenemos mayores probabilidades para cada predicción y menor cantidad de bits para representarlas. Pero al mismo tiempo la complejidad del modelo aumenta, si consideramos que hay que representar a

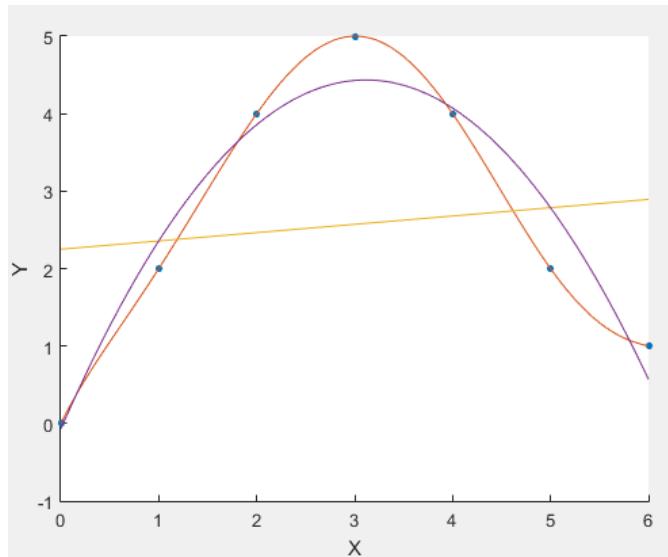


Figure 12.10: Regresión lineal, cuadrática y de orden 6

cada polinomio como un flotante de 64 bits entonces tenemos 128 bits para el modelo lineal, 384 bits para el modelo de grado 2 y 896 bits para el polinomio de grado 6. Esto quiere decir que para que el modelo de grado 6 sea mas conveniente que el lineal necesitamos que el acumulado de las diferencias en bits para las predicciones de los puntos sea de al menos 768 bits.

Observemos que cuantos mas puntos tenemos mas atractivo resulta un modelo mas complejo ya que lo que ganamos en la representación de los puntos compensa la diferencia en complejidad entre los modelos que no depende de la cantidad de puntos. Esto coincide con lo que sabemos sobre overfitting, a mayor cantidad de datos menor es la probabilidad de overfitting. El modelo MDL nos explica el motivo desde el punto de vista de la compresión de datos!

12.5.11 Clasificación con Algoritmos de Compresión

Es posible usar algoritmos de compresión en problemas de ML [Knoll-de Freitas], la distancia normalizada de compresión puede usarse como distancia para un algoritmo de clustering como K-Means. Y podemos también aplicarla a problemas de clasificación. Consideraremos por ejemplo que queremos clasificar textos en 2 clases (clasificación binaria). Podemos entonces concatenar todos los textos con label "1" del set de entrenamiento y comprimirlos. Luego concatenar todos los textos con label "0" y comprimirlos. Para clasificar podemos usar el modelo de compresión construido en cada caso y probar comprimir el nuevo texto con uno u otro modelo, el modelo que mejor comprima al texto es entonces el que usamos para predecir el label. Algo interesante es que la

diferencia de compresión nos da una idea de la certeza que tenemos y podemos calcular la probabilidad de cada clase simplemente normalizando.

Existen diferentes modelos que podemos usar al aplicar un algoritmo de compresión a un problema de clasificación:

SMDL: Standard Minimum Description Length

SMDL es el método que vimos. Creamos un modelo de compresión por cada clase y luego clasificamos en base al modelo que mejor comprime los datos nuevos. Es muy interesante reflexionar sobre la habilidad de generalizar de un modelo de compresión a partir de los datos de entrenamiento. Si los datos son muy pocos el modelo puede especializarse demasiado en ellos y luego no comprimir tan bien a nuevos datos es decir que los compresores de datos también pueden caer en overfitting y la mejor solución para evitarlo es contar con muchos datos. En este modelo el modelo de compresión no se actualiza al clasificar datos que no están en el set de entrenamiento.

AMDL: Approximate Minimum Description Length

En AMDL concatenamos al nuevo dato con su set de entrenamiento y luego calculamos la diferencia que existe entre comprimir el set de entrenamiento y el set de entrenamiento concatenado con el nuevo dato. AMDL es bastante mas lento que SMDL pero tiene como ventaja que podemos usar cualquier compresor standard ya que solo necesitamos poder comprimir archivos con el mismo mientras que en SMDL necesitamos poder modificar el código del compresor para que almacene el modelo de compresión y poder comprimir un archivo en base al modelo almacenado.

Los compresores de datos funcionan bien como clasificadores en cualquier tipo de dato que sea de tipo secuencia temporal, los archivos de texto son un buen ejemplo pero también pueden servir para series temporales y otros datos que tengan estas características. La clave es que los compresores intentan predecir en base al contenido anterior, por eso es que es importante que los datos exhiban esta propiedad. Cuando los datos no son del tipo adecuado para un compresor es posible diseñar un compresor ad-hoc y usar el mismo con alguno de los modelos vistos, por ejemplo SMDL, para clasificar.

La dualidad entre la compresión de datos y Machine Learning demuestra que comprimir implica aprender y que la capacidad de aprender está directamente relacionada con la complejidad de los datos.

BCN: Best Compression Neighbor

En BCN vamos a tomar cada archivo (o dato) del set de entrenamiento "B" y vamos a comprimir a "B" junto con el archivo que queremos clasificar y lo comparamos contra el resultado de comprimir solo el archivo que queremos clasificar. La clase asignada es la que corresponde al archivo del set de entrenamiento que mejor resultado de compresión tuvo. Esto es KNN usando la distancia de compresión como métrica.

Chapter 13

Clasificación

I think it is much more
interesting to live with
uncertainty than to live with
answers that might be wrong.

- Richard Feynman

En este capítulo vamos a estudiar distintos algoritmos de clasificación en todos los casos contaremos con un set de entrenamiento que tiene n registros en d dimensiones y con cada registro tenemos asociado un *label* que nos dice a que clase pertenece el registro. Cuando hay dos clases hablamos de clasificación binaria y cuando tenemos tres o mas hablamos de clasificación multiclasa.

El objetivo de un algoritmo de clasificación es usando el set de entrenamiento construir un modelo que permita predecir la clase de datos que no pertenecen al set de entrenamiento, es decir datos nuevos.

13.1 árboles de decisión

Un árbol de decisión es un árbol binario en donde en cada nodo dividimos el set de datos en dos de acuerdo a un cierto criterio. El objetivo es llegar a nodos hoja en los cuáles podamos clasificar correctamente nuestros datos. Vamos a usar el siguiente set de datos para la construcción del árbol.

13.1.1 ID3

Nuestro primer algoritmo es ID3 (iterative dichotomiser tree) creado por Ross Quinlan en 1979. Es un algoritmo de tipo *greedy* que en cada paso realiza el mejor split posible del set de datos en dos partes. Este proceso se repite recursivamente hasta construir el árbol final. Lo único que necesitamos para construir el árbol es determinar que split realizar en cada paso y en ID3 el criterio es seleccionar el atributo que nos da mayor *ganancia de información*.

candidato	presencia	estudios	experiencia	contratado
1	bueno	univ	alta	si
2	mala	univ	media	no
3	bueno	sec	alta	si
4	mala	univ	baja	no
5	bueno	sec	media	si
6	bueno	univ	media	si
7	regular	pri	baja	no
8	regular	univ	media	si

Table 13.1: Datos para construir un árbol de decisión.

La ganancia de información está relacionada directamente con el concepto de entropía y veremos como funciona a partir de nuestro ejemplo.

Inicialmente tenemos un total de 8 datos de los cuales 5 son contratados y tres no. Es decir que la entropía de nuestro set de datos es $H[5/8; 1/8] = 0.9544$ recordemos que esto se calcula con la fórmula: $-\sum_i P_i * \log_2 P_i$

Consideremos ahora nuestro primer atributo: **presencia**, este atributo tiene tres valores posibles: bueno, mala, regular.

presencia	contratado	no contratado
bueno	4	0
mala	0	2
regular	1	1

Table 13.2: Atributo presencia.

Para calcular la ganancia de información vamos a calcular la entropía para cada valor posible del atributo:

$$H(\text{presencia}=\text{bueno}) = H[4/4; 0/0] = 0$$

$$H(\text{presencia}=\text{mala}) = H[0/0; 2/2] = 0$$

$$H(\text{presencia}=\text{regular}) = H[1/2; 1/2] = 1$$

La entropía del atributo es el promedio ponderado de la entropía de cada valor posible por la probabilidad de que el atributo tome dicho valor. La probabilidad de "bueno" es 4/8, la probabilidad de "mala" es 2/8 y la probabilidad de "regular" es 2/8, por lo tanto la entropía de "presencia" es:

$$H(\text{presencia}) = \frac{4}{8} * 0 + \frac{2}{8} * 0 + \frac{2}{8} * 1 = 0.25$$

La ganancia de información es la diferencia entre la entropía del set de datos y la entropía del atributo:

$$GI(\text{presencia}) = 0.9544 - 0.25 = 0.7044$$

De la misma forma calcularemos la Ganancia de información de los demás atributos:

$$GI(\text{estudios}) = 0.9544 - \left(\frac{1}{8}H[0/0; 1/1] + \frac{2}{8}H[0/0; 2/2] + \frac{5}{8}H[3/5; 2/5] \right) = 0.3476$$

$$GI(\text{experiencia}) = 0.9544 - \left(\frac{2}{8}H[2/2; 0/0] + \frac{4}{8}H[1/4; 3/4] + \frac{2}{8}H[0/0; 2/2] \right) = 0.5488$$

El atributo con mayor ganancia de información es "presencia" por lo que será la raíz de nuestro árbol.

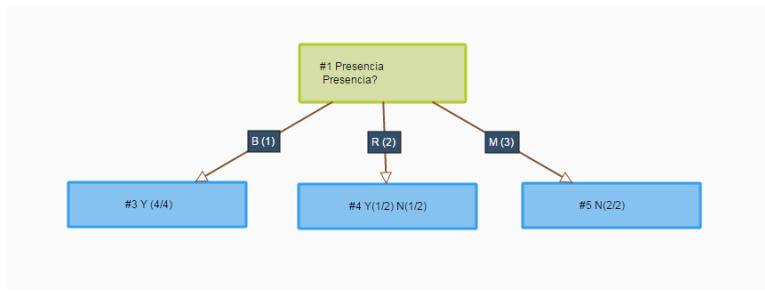


Figure 13.1: árbol de decisión ID3 primer split

Nuestro árbol ya clasifica correctamente a todos los casos de presencia buena (los contratamos) o mala (los rechazamos), nos queda el caso en el cual la presencia es regular para el cual tenemos un contratado y un rechazado. Lo que hacemos es simplemente aplicar recursivamente el algoritmo como si nuestro nodo fuera ahora nuestro set de datos, es decir que tenemos 2 datos únicamente con entropía 1 ya que tenemos probabilidad 1/2 y 1/2 para cada clase. Tenemos que elegir el atributo que mayor ganancia de información nos de. Podemos hacer esto sin cuentas porque observamos que en los dos casos en los cuales la presencia es regular podemos usar tanto los estudios como la experiencia para hacer el split.

De nuestro sencillo ejemplo podemos deducir que un árbol de decisión corre un grave peligro de caer en overfitting, en definitiva podemos hacer preguntas y preguntas hasta que cada hoja tenga únicamente nodos de una sola clase. Para evitar esto suele usarse un hiper-parámetro *minbucket* que indique la cantidad mínima de registros que puede tener un nodo no-hoja. Si usamos *minbucket*=1 caemos en overfit, por ejemplo si usamos *minbucket*=25 entonces nuestro árbol no va a particionar nodos que tengan 25 o menos registros.

Notemos también que las hojas del árbol nos dan la probabilidad de cada clase en base a cuantos registros de cada clase hay en el nodo hoja.

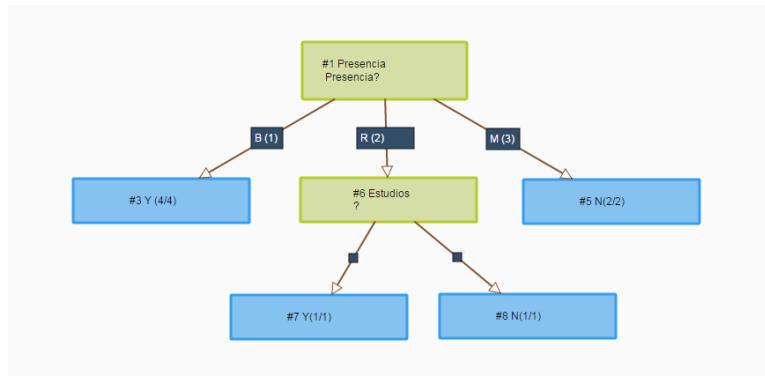


Figure 13.2: árbol de decisión ID3

13.1.2 C4.5

El algoritmo C4.5 es una mejora con respecto a ID3, es muy simple agregar las mejoras que son las siguientes:

- Aceptar atributos numéricos, para manejarlos se establece un umbral para realizar el split, por ejemplo $precio < 100.50$
- Aceptar datos con atributos faltantes. Simplemente se ignoran en el cálculo de la entropía
- Permitir que los atributos tengan un cierto peso en cuyo caso se pondera la ganancia de información por el peso (importancia) del atributo para decidir los splits
- Podar el árbol una vez creado. C4.5 examina el árbol e intenta remover las ramas que no ayudan en nada reemplazándolas con nodos hoja, esto ocurre cuando la predicción que haríamos en un nodo es igual o mejor a la predicción a la que llegamos en sus hojas.

Cuando un atributo es numérico lo que hacemos es ordenar sus valores y luego buscar linealmente cuál es el split que nos da menor entropía ya que a menor entropía mayor ganancia de información. Veamos el siguiente ejemplo:

Ordenando tenemos lo siguiente:

El mejor split lo vamos a encontrar para el valor 5, en donde nos quedan de un lado 4 de 9 registros todos con clase=1, por lo tanto su entropía es 0. Del otro lado nos quedan 5 de 9 atributos y la entropía es $\frac{5}{9}H[4/5; 1/5] = 0.4011$

Para manejar problemas de regresión podemos hacer todo exactamente como hasta ahora y en cada nodo-hoja realizar un promedio del valor de los nodos, esta será nuestra estimación para los nodos que luego lleguen hasta esa hoja.

atributo	clase
5	1
11	0
7	0
16	1
2	1
1	1
12	0
18	0
4	1

Table 13.3: Atributos numéricos.

atributo	clase
1	1
2	1
4	1
5	1
7	0
11	0
12	0
16	1
18	0

Table 13.4: Atributos numéricos.

13.2 Random Forests

El algoritmo de *Random Forests* (RF) es uno de los algoritmos mas populares en clasificación porque suele producir buenos resultados para la mayoría de los sets de datos, el gran secreto es su enorme habilidad para evitar overfitting y la fuerza de funcionar en base a un ensamble. En líneas generales un RF es una aplicación directa de *bagging* a árboles de decisión pero con una diferencia, cada árbol no usa el total de atributos sino un subset de los mismos.

En definitiva un RF es un conjunto de árboles de decisión en donde cada árbol usa un bootstrap del set de entrenamiento y un cierto conjunto de atributos tomados al azar. Los RF tienen, en general, dos hiper parámetros: la cantidad de árboles a crear y la cantidad de atributos de cada uno. El primero (cantidad de árboles) no es muy complejo ya que en general a mayor cantidad de árboles mejor funciona el ensamble pero hay que tener en cuenta que llegado un punto la mejora es prácticamente inapreciable y perdemos performance, por esto en general usaremos una cantidad de árboles que podamos manejar y que nos den el mejor resultado posible o bien que no tengan una diferencia apreciable con el uso de mayor cantidad de árboles. El segundo hiper-parámetro: la cantidad de atributos por árbol es el mas crítico y necesita ser encontrado mediante

grid-search usando la precisión OOB (out of bounds).

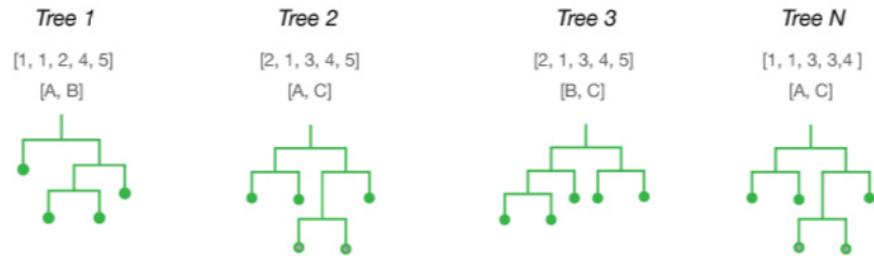


Figure 13.3: Random Forests

La figura 11.3 muestra un set de datos de 5 elementos y 3 atributos y 4 posibles árboles creados a partir del mismo. Vemos que cada árbol usa un bootstrap de 5 elementos y un conjunto al azar de 2 de los 3 atributos de nuestro set de datos.

El hecho de usar un bootstrap en lugar del set de entrenamiento completo permite evitar overfitting mientras que el usar un subconjuntos de atributos tomados al azar ayuda a determinar cuáles son los atributos que realmente resultan mas importantes para la clasificación. Un RF es capaz de darnos una medida de importancia para cada atributo promediando la precisión OOB de aquellos árboles que usan el atributo. Cuando un atributo es un buen predictor sus árboles van a tener mejores resultados que aquellos que usan un conjunto de atributos que no son buenos predictores. Con cientos o miles de árboles un RF puede tener una muy buena opinión sobre la capacidad predictora de cada atributo por lo que podemos usarlos como parte del proceso de *feature engineering* para decidir que atributos eliminar del set de datos por considerarlos ruidosos.

Los Random Forests son invariantes a la escala de los atributos es decir que no necesitamos normalizarlos. Esta es una gran ventaja con respecto a algoritmos en donde la escala es crítica como KNN o redes neuronales.

13.2.1 Distancia RF

Un RF nos permite también aprender una métrica para nuestro set de datos, si queremos calcular la distancia entre x_i y x_j lo que hacemos es clasificar a ambos puntos con nuestro RF aprendido en base al total de los puntos y luego tomar como distancia el número de árboles en los cuales la predicción para x_i y x_j es diferente. Dividiendo por el total de árboles tenemos una distancia normalizada entre 0 y 1. Es decir que dos datos para los cuales todos los árboles predicen la misma clase son idénticos de acuerdo a esta métrica. Observemos que la clase no tiene que ser la misma a lo largo de todos los árboles, lo que importa es si la clase que un árbol predice para ambos puntos es igual y no cual es esa clase o

que clase predijeron los otros árboles.

Esta distancia puede usarse luego en otros algoritmos. Las cosas empiezan a ponerse interesantes.

13.3 XGBoost

XGBoost es el algoritmo estado del arte en clasificación, una gran cantidad de competencias de ML se ganan mediante algoritmos basados en Xgboost on ensambles en donde XGboost es el clasificador base. Vamos a aprovechar esta sección para desarrollar la teoría detrás de los árboles de decisión y los ensambles basados en árboles.

Podemos generalizar la función objetivo de cualquier clasificador de la forma:

$$\text{Obj}(\Theta) = L(\Theta) + \Omega(\Theta) \quad (13.1)$$

Θ son los parámetros que el algoritmo tiene que aprender. L es el error de nuestro modelo con respecto a los datos y Ω es el factor de regularización que mide la complejidad del modelo. Al minimizar la suma buscamos un modelo que ajuste bien a los datos pero que no sea excesivamente complejo para evitar el overfitting.

La función L es:

$$L(\Theta) = \sum_{i=1}^n l(y_i, \hat{y}_i) \quad (13.2)$$

La función l es el error de predecir \hat{y}_i para y_i y depende del problema que estemos resolviendo, en un problema de regresión puede ser:

$$l = (y_i - \hat{y}_i)^2$$

En un problema de clasificación binario donde predecimos la probabilidad de que la clase sea 1.

$$l = y_i \ln(1 + e^{-\hat{y}_i}) + (1 - y_i) \ln(1 + e^{\hat{y}_i})$$

Notar que cuando $y_i = 1$ el segundo término se anula y en el primero nos queda $\ln(1 + e^{-\hat{y}_i})$ que tiende a cero a medida que \hat{y}_i tiende a 1 y tiende a infinito cuando \hat{y}_i tiende a cero. De la misma forma cuando $y_i = 0$ el primer término se anula y la función es $\ln(1 + e^{\hat{y}_i})$ que tiende a cero cuando \hat{y}_i tiende a cero y a infinito cuando \hat{y}_i tiende a 1.

13.3.1 Boosting Trees

El concepto de boosting se puede aplicar considerando que nuestra predicción se construye mediante la suma de los resultados de varios árboles.

Vamos a suponer un problema de regresión donde nuestra predicción es la sumatoria de las predicciones de varios árboles, de la forma:

$$\hat{y}_i = \sum_{j=1}^k f_j(x_i) \quad (13.3)$$

En donde f_j es el j-ésimo árbol, que pertenece a la familia de todos los árboles de regresión \mathcal{F} .

Notar que en este caso en lugar de aprender parámetros (Θ) el algoritmo está aprendiendo funciones f_j siendo cada función un árbol de regresión. Cada árbol va a intentar corregir los errores del anterior, por ejemplo si $y_i = 3$ y nuestro primer árbol predice 2.8 nuestro segundo árbol va a intentar predecir 0.2. Esta es la clave del concepto de *boosting*

De (11.2) se desprende:

$$\hat{y}_i^{(t)} = \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i) \quad (13.4)$$

Reemplazando en (11.1):

$$\text{Obj}^{(t)} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^t \Omega(f_i) \quad (13.5)$$

$$\text{Obj}^{(t)} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + \text{constant} \quad (13.6)$$

Notemos que ha quedado todo en función de f_t por lo que el problema es encontrar f_t tal que minimicemos la función objetivo. f_t es el árbol que estamos construyendo por lo que el problema se reduce a en cada paso encontrar el árbol que minimiza la función.

Reemplazando la función $l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$:

$$\text{Obj}^{(t)} = \sum_{i=1}^n (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \Omega(f_t) + \text{constant} \quad (13.7)$$

$$\text{Obj}^{(t)} = \sum_{i=1}^n [2(\hat{y}_i^{(t-1)} - y_i)f_t(x_i) + f_t(x_i)^2] + \Omega(f_t) + \text{constant} \quad (13.8)$$

Recordemos ahora la famosa serie de Taylor: $f(x + \nabla x) \approx f(x) + f'(x)\nabla x + \frac{1}{2}f''(x)\nabla x^2$

Definimos:

$$g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}) \quad (13.9)$$

Aplicando la función l de regresión:

$$g_i = \partial_{\hat{y}^{(t-1)}} (\hat{y}^{(t-1)} - y_i)^2 = 2(\hat{y}^{(t-1)} - y_i) \quad (13.10)$$

$$h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)}) \quad (13.11)$$

Aplicando la función l de regresión:

$$h_i = \partial_{\hat{y}^{(t-1)}}^2 (y_i - \hat{y}^{(t-1)})^2 = 2 \quad (13.12)$$

Reemplazando en (11.8):

$$\text{Obj}^{(t)} = \sum_{i=1}^n [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + C \quad (13.13)$$

Si removemos las constantes:

$$\text{Obj}^{(t)} = \sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \quad (13.14)$$

Con:

$$g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$$

$$h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$$

Notemos que esto sirve para cualquier función l diferenciable!

13.3.2 Complejidad de un árbol

Es necesario ahora definir la complejidad de un árbol para poder calcular Ω . En XGBoost usaremos la siguiente fórmula:

$$\Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \quad (13.15)$$

Donde T es la cantidad de hojas que tiene el árbol y w_j es el valor de la j -ésima hoja del árbol, cada hoja tiene que tener asociado un valor que es el que usamos para la regresión. γ y λ son hiper-parámetros.

En el árbol de la figura 13.4 la complejidad es: $\Omega = \gamma 3 + \frac{1}{2} \lambda (4 + 0.01 + 1)$

Cada hoja del árbol contiene un conjunto de registros (instancias) del set de datos. Vamos a definir al conjunto de instancias de la hoja j como:

$$I_j = i | q(x_i) = j \quad (13.16)$$

En donde $q(x_i)$ es una función que mapea la instancia x_i al número de hoja del árbol en la cual se encuentra la misma. Recordemos que w_j era el valor de la hoja por lo tanto $f_t(x_i) = w_{q(x_i)}$ y podemos reescribir nuestra función objetivo:

$$\text{Obj}^{(t)} = \sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \quad (13.17)$$

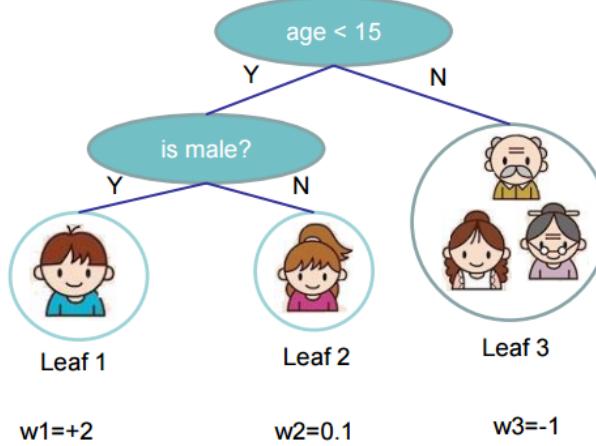


Figure 13.4: Complejidad de un árbol

$$\text{Obj}^{(t)} = \sum_{i=1}^n [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \lambda \frac{1}{2} \sum_{j=1}^T w_j^2 \quad (13.18)$$

$$\text{Obj}^{(t)} = \sum_{j=1}^T [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \quad (13.19)$$

Esto es equivalente a la suma de T funciones cuadráticas independientes.

13.3.3 Optimizando la función objetivo

Mencionemos un par de cosas sobre las funciones cuadráticas de una sola variable:

$$\operatorname{argmin}_x Gx + \frac{1}{2} Hx^2 = -\frac{G}{H}, \quad H > 0, \quad \min_x Gx + \frac{1}{2} Hx^2 = -\frac{1}{2} \frac{G^2}{H}$$

Si definimos entonces:

$$G_j = \sum_{i \in I_j} g_i \quad (13.20)$$

$$H_j = \sum_{i \in I_j} h_i \quad (13.21)$$

Volvemos a definir el objetivo:

$$\text{Obj}^{(t)} = \sum_{j=1}^T [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \quad (13.22)$$

$$\text{Obj}^{(t)} = \sum_{j=1}^T [G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2] + \gamma T \quad (13.23)$$

13.3.4 Calculando el Score de una Estructura

Si la estructura del árbol ($q(x)$) es fija entonces el peso óptimo de cada hoja y la función objetivo resultante son:

$$w_j^* = -\frac{G_j}{H_j + \lambda} \quad (13.24)$$

$$\text{Obj} = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T \quad (13.25)$$

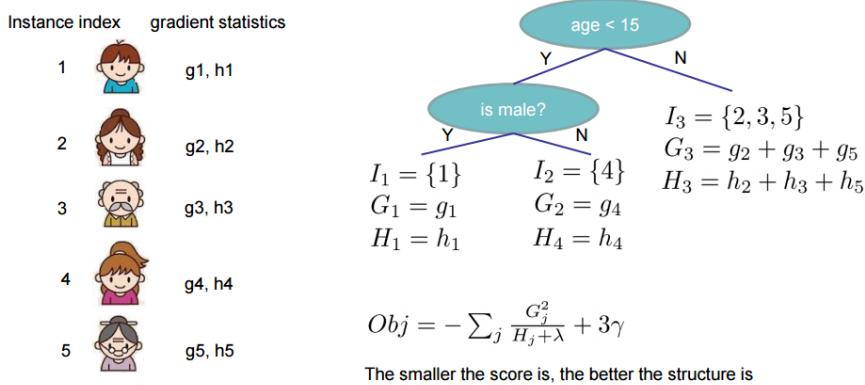


Figure 13.5: Puntaje de la estructura del árbol en XGBoost

En el ejemplo vemos que hay 5 instancias (registros), cada una con sus correspondientes estadísticas de gradiente. Vemos que la Instancia 1 está en la hoja 1, la instancia 2 en la hoja 3, la instancia 3 en la hoja 3, la instancia 4 en la hoja 2 y finalmente la instancia 5 en la hoja 3.

13.3.5 Buscando la Estructura del árbol

Tenemos ahora una función que nos permite calcular la función objetivo para una cierta estructura $q(x)$:

$$\text{Obj} = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T \quad (13.26)$$

Podríamos enumerar todas las estructuras $q(x)$ posibles y quedarnos con aquella que minimiza la función objetivo, una vez encontrada la estructura podemos calcular el peso óptimo de cada hoja como:

$$w_j^* = -\frac{G_j}{H_j + \lambda} \quad (13.27)$$

El problema es que hay demasiadas estructuras posibles y, claramente, no podemos probarlas todas. La solución en XGBoost es crecer el árbol de forma *greedy*. Empezamos con un árbol de profundidad cero, por cada nodo hoja del árbol intentamos hacer un split, el cambio en la función objetivo luego del split es:

$$\text{Gain} = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \gamma \quad (13.28)$$

El primer término es el puntaje del hijo izquierdo, el segundo término el puntaje del hijo derecho y el tercer término es el puntaje si no hacemos el split. γ representa el costo de agregar otra hoja al árbol.

Para determinar de que forma hacer el split lo que hacemos es ordenar los valores del atributo y y luego buscar de izquierda a derecha el split que nos de mejor ganancia para ese atributo. Repetimos esto para cada atributo y elegimos el mejor de todos los splits que hemos analizado, es decir que atributo usar y por que valor hacer el split. Las variables categóricas se convierten en k variables binarias usando one-hot-encoding, en una variable binaria el split a considerar es siempre binario (1 o 0).

13.3.6 XGBoost en la práctica

Resumimos aquí el funcionamiento del algoritmo:

- Agregamos un árbol en cada iteración
- Al principio de cada iteración calculamos $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ y $h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$
- Usamos estas estadísticas para hacer crecer un árbol $f_t(x)$ de forma greedy

$$\text{Obj} = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$

- Agregamos $f_t(x)$ al modelo $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x)$ aunque en general lo que hacemos es $y^{(t)} = y^{(t-1)} + \epsilon f_t(x)$ en donde ϵ es un factor en el orden de 0.1. Esto quiere decir que no hacemos una optimización completa en cada paso sino que reservamos algo para las rondas siguientes, esto previene el overfitting (!)

Si bien no tiene sentido implementar XGBoost desde cero es muy recomendable haber entendido el algoritmo para saber que es lo que estamos haciendo y que significan los diferentes hiper-parámetros cuando lo usamos. Existen implementaciones de XGBoost muy eficientes en varios lenguajes y es uno de los algoritmos de clasificación y regresión que mejores resultados obtiene, podríamos decir que XGBoost es para clasificar lo que PAQ es para comprimir. No siempre es el mejor pero casi siempre da buenos resultados y es siempre recomendable probarlo.

13.4 Naïve Bayes

Este algoritmo está basado en el teorema de Bayes, tiene varias propiedades interesantes. Es un algoritmo extremadamente sencillo, rápido y que escala muy bien, casi nunca es "el mejor" algoritmo pero tampoco suele ser el peor en la gran mayoría de los problemas. Ha sido usado con éxito en varias aplicaciones y debe su fama a los primeros filtros de Spam exitosos que estaban basados en este algoritmo.

El único elemento necesario para entender el algoritmo es el teorema de Bayes así que vamos a explicarlo directamente mediante un ejemplo. Supongamos que tenemos un set de datos de 8 documentos sobre deportes clasificados en tres clases: fútbol, ajedrez y polo.

```
D1 (polo) = pelota equipo caballo handicap equipo
D2 (ajedrez) = jaque caballo elo
D3 (futbol) = pelota equipo foul deporte pelota
D4 (ajedrez) = deporte equipo elo
D5 (futbol) = foul pelota equipo pelota
D6 (futbol) = deporte pelota pelota
D7 (polo) = caballo foul caballo deporte
D8 (futbol) = caballo foul foul
```

Podemos rápidamente construir una matriz con la cantidad de veces que aparece cada palabra en cada clase de la forma:

Esta tabla nos indica rápidamente la probabilidad de cada término en su clase, por ejemplo la probabilidad de "pelota" en polo es 1/9 ya que la suma de todas las frecuencias de todos los términos en la clase polo es 9 y pelota aparece 1 vez.

Lo que queremos calcular es, dado un documento nuevo, cual es la probabilidad de cada una de las tres clases: $P(\text{polo}|D)$, $P(\text{ajedrez}|D)$, $P(\text{fútbol}|D)$. Si podemos hacer esto entonces simplemente clasificamos al documento de acuerdo a que clase es mas probable. Aplicando el teorema de Bayes:

$$P(\text{class}|D) = \frac{P(\text{class})P(D|\text{class})}{P(D)} \quad (13.29)$$

La probabilidad de cada clase es simplemente la cantidad de documentos de dicha clase sobre el total de documentos, En nuestro caso $P(\text{polo})=2/8$,

Término	Polo	Ajedrez	Fútbol
pelota	1	0	6
equipo	2	1	2
caballo	3	1	1
handicap	1	0	0
jaque	0	1	0
elo	0	2	0
foul	1	0	4
deporte	1	1	2
total	9	6	15

Table 13.5: Ejemplo Naïve Bayes.

$P(\text{ajedrez})=2/8$ y $P(\text{futbol})=4/8$. La probabilidad de cada documento la podemos suponer 1, en realidad no nos interesa porque vamos a comparar la probabilidad del documento en cada clase y el denominador es siempre el mismo así que podemos ignorarlo (el denominador no depende de la clase).

Por lo tanto si podemos resolver $P(d\text{---class})$ tenemos todo el algoritmo resuelto.

La probabilidad de un documento dada una clase depende de las palabras que aparezcan en el documento y la probabilidad de estas palabras en dicha clase, si suponemos que las palabras de un documento son independientes podemos calcular:

$$P(D|\text{class}) = \prod_{w_i \in D} P(w_i|\text{class}) \quad (13.30)$$

Donde $P(w_i\text{---class})$ es simplemente la cantidad de veces que la palabra aparece en la clase dividido la suma total de todas las frecuencias de todas las palabras en la clase. Para evitar underflow en lugar de multiplicar las probabilidades podemos sumar el logaritmo de las mismas:

$$P(D|\text{class}) = \sum_{w_i \in D} \log(P(w_i|\text{class})) \quad (13.31)$$

Por supuesto que suponer que las palabras de un texto son independientes no es algo correcto, pero para Bayes esto funciona bien, el modelo solo tiene en cuenta si la palabra aparece o no en el documento y nada más, por esto se lo llama *naïve*, por basarse en una suposición que es muy inocente.

13.4.1 El Problema de las probabilidades cero

Supongamos que tenemos $D=\text{pelota, barrilete}$. La palabra barrilete no ocurre en ninguna de nuestras clases por lo tanto $P(\text{barrilete---class})$ es igual a cero para cualquier clase. Por lo tanto independientemente de las otras probabilidades vamos a decir que $P(D\text{---class})$ es cero para todas las clases. Esto claramente está mal.

La solución pasa por aplicar la corrección de Laplace que ya vimos al estudiar modelos de lenguajes. Simplemente suponemos que la frecuencia de palabras nuevas es 1 y al cálculo de las probabilidades siempre le agregamos la cantidad de palabras del vocabulario. Esto es equivalente a sumar 1 a las frecuencias de todas las palabras. Por ejemplo calculemos $P(\text{handicap—ajedrez})$ vemos que la frecuencia de handicap en ajedrez es cero por lo tanto vamos a calcular $P(\text{handicap—ajedrez}) = 1/(6 + 8) = 0.071$. Para palabras que ya existen hay que hacer lo mismo, por ejemplo $P(\text{pelota—polo}) = 2/9 + 8 = 0.118$. Si una palabra no existe en todo el vocabulario hay que tener cuidado de sumar 1 a la longitud del vocabulario. Por ejemplo $P(\text{barrilete—polo}) = 1/(9+9)$. Ya que ahora no tenemos 8 palabras en total sino 9. La corrección de Laplace es muy sencilla y permite calcular la probabilidad de un documento dada una clase de forma casi automática.

13.4.2 Bayes en acción

A modo de ejemplo clasifiquemos D=deporte,equipo, caballo.

$$\begin{aligned} P(\text{polo—deporte,equipo,caballo}) &= P(\text{polo}) P(\text{deporte—polo}) P(\text{equipo—polo}) \\ P(\text{caballo—polo}) &= 0.25 * 0.176 * 0.235 * 0.118 = 0.00121 \end{aligned}$$

$$\begin{aligned} P(\text{ajedrez—deporte,equipo,caballo}) &= P(\text{ajedrez}) P(\text{deporte—ajedrez}) P(\text{equipo—ajedrez}) \\ P(\text{caballo—ajedrez}) &= 0.000729 \end{aligned}$$

$$\begin{aligned} P(\text{futbol—deporte,equipo,caballo}) &= P(\text{fútbol}) P(\text{deporte—fútbol}) P(\text{equipo—fútbol}) \\ P(\text{caballo—fútbol}) &= 0.000740 \end{aligned}$$

Por lo tanto clasificaríamos nuestro documento como "polo".

En este ejemplo multiplicamos directamente las probabilidades porque los datos son pocos, recordemos que en lugar de hacer esto hay que sumar el logaritmo de las probabilidades.

Bayes puede ser víctima de overfitting. Si existe una palabra rara que solo aparece en una clase y en ninguna otra entonces cada vez que Bayes vea esta palabra va a clasificar automáticamente al documento dentro de esa clase, el algoritmo alucina que cualquier cosa que tiene esa palabra pertenece a la única clase en la que la observó. Una posible solución es ignorar las palabras que tienen una frecuencia baja o que solo se observaron en una clase. En la práctica esto a veces es necesario y a veces no.

Bayes es un clasificador muy popular para textos pero puede usarse también en cualquier otro caso, su principal virtud es la capacidad de escalar a datos realmente masivos sin ningún tipo de problema, no todos los algoritmos tienen esta ventaja.

13.5 Perceptrón

Perceptrón es un algoritmo de clasificación lineal. El algoritmo surge a fines de la década del 50 y fue desarrollado por Frank Rosenblatt . En aquel entonces se buscaba crear un algoritmo que imitara el funcionamiento del cerebro humano. La teoría era que una neurona podía programarse como una salida

binaria que dependía del resultado de la combinación lineal entre los inputs y pesos asignados a los mismos. La neurona respondía 0 o 1 según el resultado de la combinación lineal fuese mayor o menor a un cierto umbral.

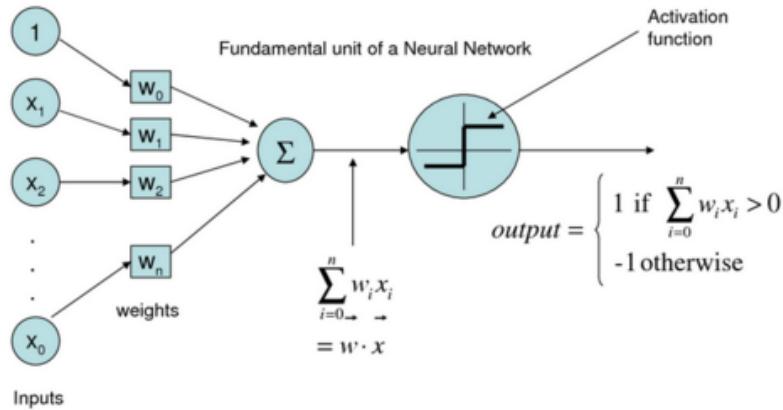


Figure 13.6: Perceptrón

El diagrama muestra una típica forma de representar un perceptrón, notar que hay un “input” extra con valor 1 para el desplazamiento “b” de la función lineal w^*x+b . La función de activación debe emitir un resultado binario que puede ser 1 o 0 , -1 o 1 etc. Para decidir que resultado emitir entre los dos posibles la función de activación, en este caso, emite 1 si w^*x es mayor a cero o 0 en caso contrario. Es posible usar cualquier otro tipo de función de activación siempre y cuando sea una función que reciba un número y devuelva dos valores posibles.

Definido de esta forma es fácil darse cuenta que perceptrón puede usarse para clasificación binaria, en sus primeras versiones se usó en un robot que contaba con células fotoeléctricas como inputs y podía de esta forma clasificar un objeto que estuviera delante del mismo, en su momento éste fue un avance revolucionario para la entonces joven rama de la computación conocida como Inteligencia Artificial, sin embargo debe quedar claro que Perceptrón es un algoritmo que encuentra un hiperplano separador y no mas que esto por lo que dista bastante de emular cualquier cerebro.

13.5.1 Algoritmo Base

El algoritmo mediante el cual perceptrón “aprende” los valores de “w” es extremadamente simple:

- El Perceptrón base usa -1 y 1 como clases.
- La predicción se hace calculando $w * x$

- La función de activación es la función signo, es decir -1 si $w \cdot x$ es menor a 0 o 1 en caso contrario (el cero se define positivo por convención).
- La regla de actualización es: $w = w + x * y$ que solo aplicamos cuando nuestra predicción falla. Notemos que y nos da el signo para la actualización de w .

$$w = w + y_i x_i \quad (13.32)$$

Algorithm 22: Perceptrón base

```

1 while not fin do
2   |  tomar un punto y su clase  $(x_i, y_i)$ 
3   |  if  $\text{sign}(w \cdot x) \neq y$  then
4     |    |   $w = w + y * x$ 

```

13.5.2 Algoritmo Mejorado

En el algoritmo mejorado agregamos un factor de aprendizaje para ir actualizando w de forma gradual y además permitimos que y tome dos valores cualesquiera y podemos usar cualquier función de activación.

Algorithm 23: Perceptrón mejorado

```

1 while not fin do
2   |  tomar un punto y su clase  $(x_i, y_i)$ 
3   |  if  $f(w \cdot x) \neq y$  then
4     |    |   $w = w + \alpha * y * x$ 

```

$$w = w + \alpha y_i x_i \quad (13.33)$$

Si por ejemplo usamos $y = 0, 1$ Es decir que cuando $y == 0$ y nuestra predicción fue 1 actualizamos $w = w - \alpha * x$ y cuando $y == 1$ pero nuestra predicción fue 0 $w = w + alpha * x$

Debemos notar varias cosas interesantes. Primero y principal es un algoritmo online! Procesa los puntos uno por uno y nunca necesita memoria para mas de un punto a la vez. En segundo lugar los pesos solo se actualizan si el algoritmo está clasificando mal el punto, si la predicción es correcta el algoritmo no hace nada. El algoritmo termina cuando la cantidad de errores es cero o bien cuando se han realizado la cantidad de iteraciones que deseamos hacer.

El costo de perceptrón está representado por la cantidad de errores que comete, es decir la cantidad de puntos que no puede clasificar correctamente en el set de entrenamiento.

13.5.3 Normalización Online

En algunos casos es conveniente para que la convergencia sea más rápida normalizar todas las columnas de nuestros datos para que tengan promedio cero y desviación standard 1. Sin embargo esto no puede hacerse en un algoritmo online o que procese streams porque no conocemos la totalidad de los datos. Una forma de simular la normalización de los datos es usar $1 + \log(x)$ en lugar de x , es decir tomar $1 +$ el logaritmo del dato para la columna que nos interese. De esta forma si una columna de nuestros datos tiene una escala completamente diferente a otra la diferencia se reduce notablemente por el efecto del logaritmo.

13.5.4 Predicciones

Una vez que perceptrón ha aprendido la recta separadora es muy simple clasificar cualquier punto nuevo, simplemente multiplicamos por w (a veces lo llamamos θ) y aplicamos al resultado la función de activación.

Una de las características mas importantes de Perceptrón es su capacidad de encontrar siempre una separación lineal, si es que existe y converger rápidamente a una solución. Vamos a demostrar esto estudiando el teorema de convergencia del algoritmo.

13.5.5 Convergencia

El teorema de convergencia de perceptrón dice: "si las clases son linealmente separables perceptrón converge en un número finito y acotado de pasos".

Este teorema es fundamental pues asegura que teniendo clases linealmente separables eventualmente perceptrón va a encontrar alguna separación entre las mismas produciendo cero errores en la clasificación del set de entrenamiento.

Notar que existen infinitas líneas que separan las clases. De todas estas perceptrón va a encontrar una cualquiera, hay que analizar si esto puede ser o no un problema.

El ejemplo muestra los distintos hiperplanos que el algoritmo prueba hasta encontrar una separación perfecta entre las clases.

Theorem 26 (Block-Novikoff) *Sea un conjunto de m puntos y sus correspondientes labels: $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$. Donde $y_i = \pm 1$ y supongamos que $\|x_i\| \leq D$. Sea u un vector unitario ($\|u\| = 1$) tal que $y_i(x_i u) \geq \gamma$. Es decir que u separa a las clases ya que cuando $y = 1$ $xu \leq \gamma$ y cuando $y = -1$ $xu \leq -\gamma$.*

Entonces el número total de errores que Perceptrón comete antes de converger es a lo sumo $(D/\gamma)^2$

Para demostrar el teorema vamos a usar el algoritmo base de perceptrón en donde si $x\theta \geq 0$ predecimos 1 y si $x\theta < 0$ predecimos -1.

No vamos a usar factor de aprendizaje por lo que la regla de actualización será: $\theta = \theta + yx_i$

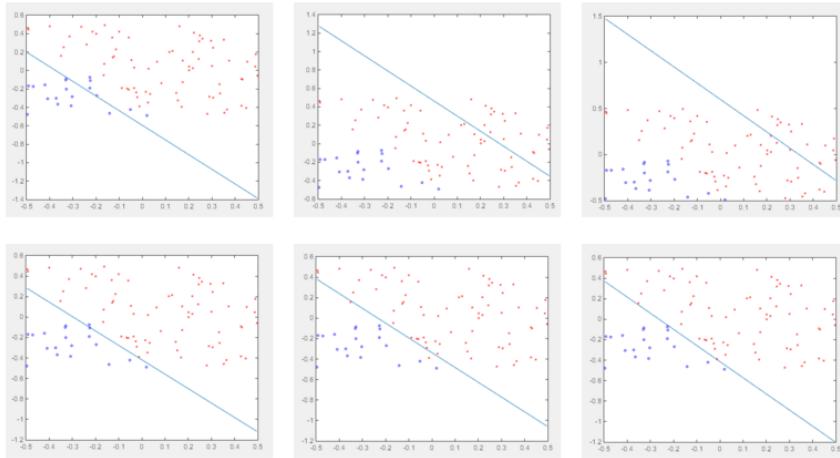


Figure 13.7: Convergencia de Perceptrón

Sabemos que perceptrón solo actualiza el hiperplano θ cada vez que comete un error. Sea $\theta^{(k)}$ el valor de θ anterior a que el algoritmo cometa el k -ésimo error. Comenzamos con $\theta^{(1)} = \vec{0}$ que es la inicialización del algoritmo.

Si el error k ocurre con el dato (x_i, y_i) Entonces $\text{sign}(x_i\theta^{(k)}) \neq y_i$, por lo tanto $x_i\theta^{(k)} \leq 0$

Además por la regla de actualización sabemos que $\theta^{(k+1)} = \theta^{(k)} + y_i x_i$

Entonces tenemos que: $u\theta^{(k+1)} = u\theta^{(k)} + y_i x_i u$ y por lo tanto $u\theta^{(k+1)} \geq u\theta^{(k)} + \gamma$.

Por inducción $u\theta^{(k+1)} \geq k\gamma$

Por otro lado:

$$\begin{aligned} \|\theta^{(k+1)}\|^2 &= \|\theta^{(k)} + y_i x_i\|^2 \\ &= \|\theta^{(k)}\|^2 + \|x_i\|^2 + 2y_i x_i \theta^{(k)} \\ &\leq \|\theta^{(k)}\| + \|x_i\|^2 \quad (x_i \theta^{(k)} \leq 0) \\ &\leq \|\theta^{(k)}\| + D^2 \end{aligned} \tag{13.34}$$

Aplicando inducción directa:

$$\|\theta^{(k+1)}\|^2 \leq kD^2$$

Combinando $\|\theta^{(k+1)}\|^2 \leq \|\theta^{(k)}\| + D^2$ y $u\theta^{(k+1)} \geq k\gamma$ tenemos:

$$\sqrt{k}D \geq \|\theta^{(k+1)}\|$$

Sabemos que $\|u\| = 1$ y que $zu = \|z\|\|u\| \cos \phi$ por lo tanto:

$$\sqrt{k}D \geq \theta^{(k+1)}u$$

Recordemos: $u\theta^{(k+1)} \geq k\gamma$

$$\sqrt{k}D \geq k\gamma u$$

$$k \leq (D/\gamma)^2$$

Que es lo que queríamos demostrar.

13.5.6 Funciones de Activación

A continuación mostramos algunas funciones de activación típicas que pueden usar con perceptrón.

Signo

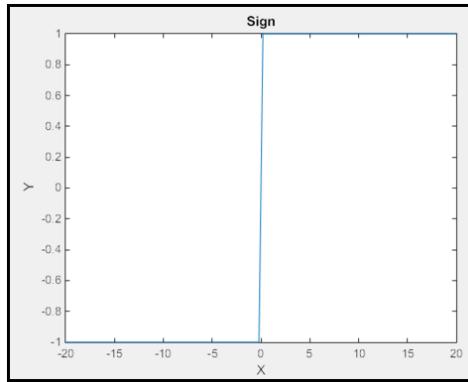


Figure 13.8: Función signo

$$f(x) = \text{sign}(x) \quad (13.35)$$

La función signo es la mas simple de todas, simplemente devuelve 1 si $X > 0$ o -1 si $X < 0$ o 0 si $X = 0$. La función signo tiene derivada 0 en todos los puntos excepto en el 0 en donde no es diferenciable.

Sigmoid

$$f(x) = \frac{1}{1+e^{-x}} \quad (13.36)$$

La función sigmoidea es muy usada para convertir valores a un número entre 0 y 1. Todos los valores menores a 0 generan un número cercano a 0 y todos los valores mayores a 0 generan un número cercano a 1. Cuanto mas chico x mas parecido a 0 el resultado y cuanto mas grande x mas parecido a 1 el resultado.

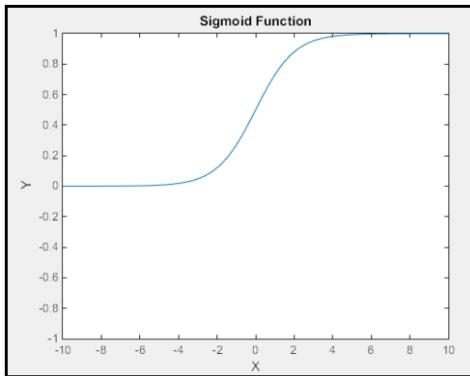


Figure 13.9: Función Sigmoidea

El resultado de la función sigmoidea al ser un número entre 0 y 1 puede considerarse una probabilidad. Por ejemplo la probabilidad de que un cierto punto pertenezca a una clase.

La derivada de la función sigmoidea es muy sencilla:

$$f'(x) = f(x)(1 - f(x)) \quad (13.37)$$

Si hemos calculado previamente $f(x)$ podemos calcular su derivada usando simplemente una resta y un producto sin necesidad de volver a calcular la exponenciación. Esto hace que la función sigmoidea sea eficiente en algoritmos que usan el gradiente de funciones de múltiples variables como por ejemplo Gradient Descent.

Tanh

$$f(x) = \frac{\sinh(x)}{\cosh(x)} \quad (13.38)$$

La tangente hiperbólica es otra función muy usada como función de activación. Genera un número entre -1 y 1. Es muy similar a la función sigmoidea pero con un rango mayor para su resultado.

La derivada de la función tanh también es muy simple.

$$f'(x) = 1 - f(x)^2 \quad (13.39)$$

Como podemos ver al igual que con la función sigmoidea una vez que hemos calculado la función de activación podemos calcular su derivada usando simplemente una resta y un producto. Es de destacar que la función tanh tiene un gradiente más pronunciado que la función sigmoidea por lo que será mas robusta en algoritmos en los cuales es necesario propagar el gradiente en diferentes etapas.

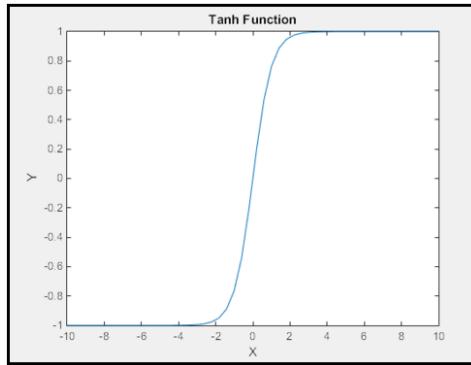


Figure 13.10: Función Tanh

13.5.7 Perceptrón Multiclasé

Perceptrón es por definición un clasificador binario pero podemos adaptarlo o extenderlo a problemas de clasificación multiclasé, es decir con n clases posibles. Hay muchas alternativas para lograr esto pero vamos a mencionar las dos opciones más típicas que no solo se aplican a Perceptrón sino a cualquier otro clasificador binario que queramos extender a un problema con más de dos clases.

One vs All

En one vs all vamos a entrenar perceptrón por cada clase (1 pertenece y 0 no pertenece) y luego cuando queremos clasificar un dato nuevo aplicamos todos los vectores aprendidos, el resultado mayor nos indica a qué clase pertenece el dato.

$$c = \arg \max_y w_y x$$

Notemos que no aplicamos la función de activación sino que simplemente dejamos expresado el resultado de $w * x$ y luego teniendo todos los resultados tomamos el mayor para decidir la clase.

One vs One

En el método one vs one vamos a aplicar un Perceptrón binario a cada par de clases posibles, si tenemos 4 clases vamos a usar un Perceptrón para la clase 1 y la 2, otro para la 2 y la 3, otro para 3 y 4, otro para 2 y 3 y el último para 3 y 4. Con todos estos clasificadores cada vez que queremos clasificar un dato aplicamos los diferentes perceptrones y luego elegimos la clase que más “triunfos” tuvo. Por ejemplo con 4 clases si nuestro dato x produce como resultados $c_1 < c_2, c_1 < c_3, c_1 > c_4, c_2 > c_3, c_2 > c_4, c_3 > c_4$ tenemos que en

total la clase 1 ganó una vez, la clase 2 ganó 3 veces, la clase 3 ganó 2 veces y la clase 4 ganó 0 veces. Por lo tanto clasificaríamos a x como clase 2.

One vs One en general es algo más robusto que One vs All pero requiere de mayor cantidad de recursos, si las clases son muchos One vs One tiene que construir una cantidad combinatoria de clasificadores y es inviable por lo que optaríamos por One vs All.

13.5.8 Conclusiones

Perceptrón es un clasificador lineal extremadamente simple. El algoritmo funciona, por definición de forma online pudiendo procesar streams o datos masivos sin problemas. Segundo lo que hemos experimentado este algoritmo es particularmente bueno para procesar datos binarios (0s y 1s) muy dispersos en un espacio dimensional enorme. Es decir que es una excelente opción para clasificar textos de forma muy eficiente.

El algoritmo fue olvidado durante muchos años cuando los investigadores se dieron cuenta que no cumplía con su promesa de simular el comportamiento del cerebro humano, muchos años después este algoritmo renace por su enorme eficiencia como clasificador lineal. Aunque tenga muchos años Perceptrón sigue siendo una opción a considerar y es la base de otros algoritmos muy usados con las redes neuronales o SVM.

13.6 SVM

Bienvenidos al gran monstruo de los clasificadores lineales: SVM. Un algoritmo que es muy usado para resolver todo tipo de problemas y que tiene muchísimas versiones y variantes. Hoy en día SVM es una de las herramientas más usadas y más poderosas para resolver problemas de clasificación.

SVM es otro algoritmo de clasificación lineal, es decir que la salida del mismo será un vector n-dimensional. Una recta en dos dimensiones o un hiperplano en n-dimensiones. La idea es muy simple y podemos expresarla a partir de lo que sabemos de nuestro querido Perceptrón. En Perceptrón buscábamos algún hiperplano que separase las clases que queremos clasificar. En SVM el objetivo es encontrar el mejor hiperplano.

En SVM el mejor hiperplano es aquel que maximiza el margen entre las clases. Es decir el que logra la máxima separación entre las clases. De todos los separadores lineales posibles hay solo uno que cumple con esta condición.

En un alarde de imaginación vamos a empezar por el principio estableciendo la notación que vamos a usar que es parecida pero no exactamente igual a la notación que hemos usado con los algoritmos anteriores, una verdadera maldad.

Tenemos m datos en n dimensiones representados en una matriz de $m \times n$. Cada punto lo identificamos como x_i . Tenemos 2 clases que vamos a representar con -1 y 1. El vector y de longitud m tiene la clase a la que pertenece cada punto.

Vamos a partir del planteo de Perceptrón:

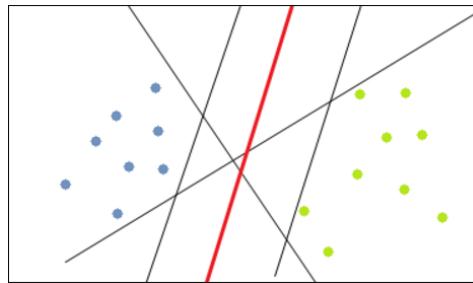


Figure 13.11: SVM

$$f(w, b) = g(x * w + b) \quad (13.40)$$

g es la función de activación, que en este caso es 1 si $x * w + b$ es mayor a cero o -1 en caso contrario. Hasta aquí tenemos Perceptrón en su versión básica pero desdoblando el desplazamiento del vector w , aquí no agregamos la columna de unos porque el desplazamiento en SVM requiere de un cuidado especial. Notar que si $xw + b$ es un hiperplano entonces w es un vector perpendicular al mismo ya que la ecuación del hiperplano es $xw + b = 0$. Si no recuerdan esto por favor hagan un ejemplo en el plano para convencerse de que esto es cierto por ejemplo si $w_1 = 1, w_2 = -1$. El plano es $w_1x + w_2y = 0$ es decir $x - y = 0$, esta es la recta $x = y$ a la cual $w = (1, -1)$ es perpendicular.

Lo que vamos a agregar para ir transformando Perceptrón en SVM es el concepto de “margen”.

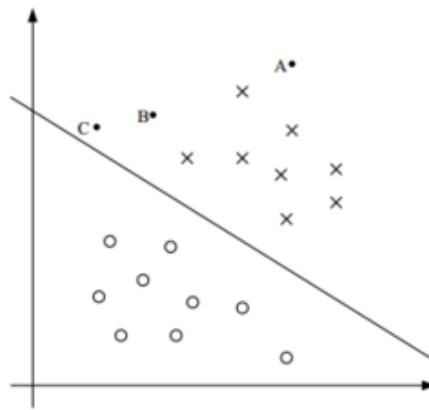


Figure 13.12: SVM

La idea del margen es muy simple. En el diagrama vemos tres puntos: A,B y C. Por su ubicación sabemos que $A * w + b$ es mayor que $B * w + b$ que a su vez es mayor que $C * w + b$. Cuanto mayor sea el “margen” más seguros vamos a estar sobre la clasificación del punto. En este caso estamos totalmente seguros de clasificar a “A” como una cruz pero con respecto a “C” nuestro nivel de confianza es menor, ya que un mínimo desplazamiento del hiperplano que separa las clases podría hacer que “C” quede del otro lado del mismo y lo clasifiquemos dentro de la clase “circulito”.

Por lo tanto vamos a decir que el margen de A es mayor que el de B y éste a su vez es mayor que el de C. Vamos a definir el margen de dos formas diferentes.

13.6.1 Margen Funcional

Dado un punto x_i y su clase y_i , un vector w y un desplazamiento b . Definimos el margen de x_i como $\hat{\gamma}_i$.

$$\hat{\gamma}_i = y_i(w * x_i + b) \quad (13.41)$$

Es muy fácil interpretar este margen. Cuando la clase es +1 el margen es grande si $w * x + b$ es un número grande y cuando la clase es -1 el margen es grande cuando $w * x + b$ es un número muy negativo.

Vamos a tomar como margen funcional de nuestro conjunto de datos al menor de todos los márgenes funcionales:

$$\hat{\gamma} = \arg \min_i \hat{\gamma}_i \quad (13.42)$$

El margen funcional es muy claro pero tiene un problema: podemos hacer crecer a w y b tanto como queramos para que el margen sea cada vez mas grande.

Debe quedar claro que si escalamos w y b multiplicándolos por una constante en realidad no estamos clasificando mejor, de hecho estamos clasificando igual que antes porque $g(cwx + cb)$ va a seguir siendo positivo en los casos en que $g(xw + b)$ era positivo y negativo en los demás casos. Es decir que podríamos llevar al margen funcional a números arbitrariamente grandes sin estar clasificando mejor. Esto lo tenemos que corregir.

13.6.2 Margen Geométrico

Queremos ahora calcular el margen geométrico de “A” que es el vector AB . El punto “A” está representado por x_i , Y sabemos que $w/\|w\|$ es el vector unitario que tiene igual dirección que w . Podemos entonces proyectar γ_i sobre w calculando $\gamma_i * \frac{w}{\|w\|}$. Una vez que hacemos esto es sencillo calcular el punto “B” como: $x_i - \gamma_i \frac{w}{\|w\|}$. Este punto “B” reside en el hiperplano separador por lo tanto: $(x_i - \gamma_i \frac{w}{\|w\|})w + b = 0$ y de aquí podemos despejar γ_i .

$$\gamma_i = \frac{x_i w + b}{\|w\|} = x_i \left(\frac{w}{\|w\|} \right) + \frac{b}{\|w\|} \quad (13.43)$$

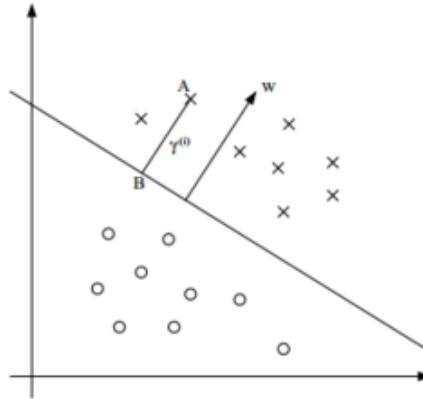


Figure 13.13: SVM

Este es el margen geométrico y lo llamamos γ_i para distinguirlo del margen funcional $\hat{\gamma}_i$. Notar que vale para los puntos a uno u otro lado del hiperplano separador.

Observemos dos cosas muy importantes sobre el margen geométrico: En primer lugar es invariante a escalar w y b por una constante que es lo que buscábamos. En segundo lugar si $\|w\| = 1$ entonces el margen geométrico y el margen funcional son iguales. Como hemos dicho que podíamos escalar a w arbitrariamente vamos a pedir que $\|w\| = 1$ lo cual nos simplifica mucho las cosas.

Al margen geométrico global lo definimos como el mínimo de todos los márgenes geométricos:

$$\gamma = \arg \min_i \gamma_i \quad (13.44)$$

Nuestro objetivo es maximizar la separación entre clases, es decir maximizar el margen mínimo. Podemos escribir entonces el siguiente planteo:

$$\begin{aligned} & \max_{w,b} \gamma \\ & \text{s.t.: } y_i(w * x_i + b) \geq \gamma, i = 1..m \\ & \quad \|w\| = 1 \end{aligned} \quad (13.45)$$

Notar que necesitamos $\|w\| = 1$ para que las restricciones no involucren a $\|w\|$. Tener también presente que es el mínimo de todos los márgenes es decir que estamos maximizando el margen mínimo.

Recordemos que $\gamma = \frac{\hat{\gamma}}{\|w\|}$ por lo que podemos escribir el mismo problema maximizando el márgen funcional:

$$\begin{aligned} & \max_{w,b} \frac{\hat{\gamma}}{\|w\|} \\ & st : y_i(w * x_i + b) \geq \gamma \quad , i = 1..m \end{aligned} \tag{13.46}$$

Cambiando el márgen geométrico por el funcional no hemos modificado en nada el problema. Y recordemos que podemos escalar a w y b por un factor constante sin cambiar el problema, vamos a buscar entonces valores para w y b tales que $\hat{\gamma} = 1$ con lo cual nuestro problema es:

$$\begin{aligned} & \max_{w,b} \frac{1}{\|w\|} \\ & st : y_i(w * x_i + b) \geq \gamma \quad , i = 1..m \end{aligned} \tag{13.47}$$

Maximizar $\frac{1}{\|w\|}$ es lo mismo que minimizar $\|w\|$ por lo que podemos escribir el problema como:

$$\begin{aligned} & \min_{w,b} \|w\| \\ & st : y_i(w * x_i + b) \geq \gamma \quad , i = 1..m \end{aligned} \tag{13.48}$$

Y como es lo mismo por conveniencia vamos a multiplicar a $\|w\|$ por $1/2$ y a deshacernos de la raíz cuadrada en la norma .

$$\begin{aligned} & \min_{w,b} \frac{1}{2} \|w\|^2 \\ & st : y_i(w * x_i + b) \geq \gamma \quad , i = 1..m \end{aligned} \tag{13.49}$$

Este es un problema de programación lineal cuadrática ya que y_i, x_i son valores constantes y el funcional es una función cuadrática. Podemos resolverlo por ejemplo usando algún solver de PLC (programación lineal cuadrática) y ahí terminaría el problema. Sin embargo vamos a desarrollar un algoritmo para poder resolverlo nosotros mismos.

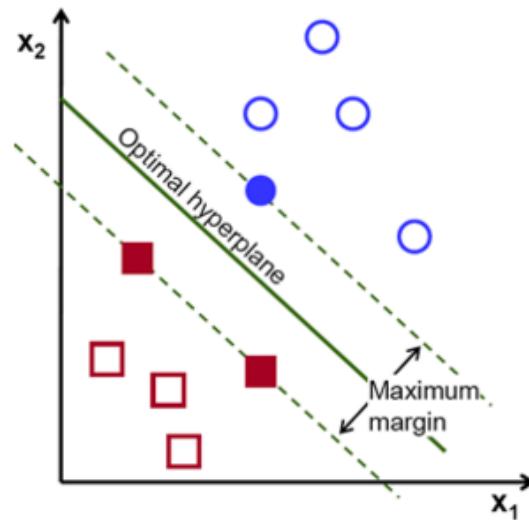


Figure 13.14: SVM

13.6.3 Support Vectors

Veamos nuevamente un vistazo a la geometría de nuestro problema simplificado al plano:

Algo muy interesante que tenemos que observar es que SVM se basa en encontrar el hiperplano para el cual el margen de separación mínimo es máximo. Esto implica que algunos de nuestros puntos van a quedar sobre los márgenes, estos puntos son los que llamamos “Support Vectors” y son los que definen el nombre del algoritmo. Es muy importante destacar que SVM solo necesita de estos puntos para definir el hiperplano separador y todos los demás puntos los podríamos ignorar completamente!

Tenemos que mencionar que en general la cantidad de support vectors debería ser cercana a la cantidad de dimensiones de los datos “n” +1. Por ejemplo en dos dimensiones hacen falta tres support vectors. Dos de una clase para definir uno de los márgenes y el tercero es un punto de la otra clase por la que pasa una recta paralela a la primera.

Interpretaremos primero nuestro planteo final, en definitiva hemos llegado a la conclusión de que SVM se resume a minimizar la norma del vector w que es perpendicular al hiperplano separador. Con esto logramos maximizar el margen mínimo entre un punto cualquiera de nuestro set de datos y el hiperplano separador. En concreto imaginemos que tenemos dos puntos, uno x^+ con clase +1 y otro x^- con clase -1 que residen en los márgenes, es decir que son support vectors. La distancia entre los mismos es $x^+ - x^-$

Esta distancia la queremos maximizar.

Y podemos calcular el margen que separa a los puntos proyectando sobre w :

$$(x^+ - x^-) \frac{w}{\|w\|} \quad (13.50)$$

Sabemos que si los puntos están en los márgenes entonces $wx^+ + b = 1$ y $wx^- + b = -1$ entonces $wx^+ = 1 - b$ y $wx^- = -1 - b$ Reemplazando:

$$(1 - b - (-1 - b)) \frac{1}{\|w\|} = \frac{2}{\|w\|} \quad (13.51)$$

Con lo cual explicamos que maximizar la distancia equivale a maximizar $1/\|w\|$ que equivale a minimizar $\|w\|$ o $\frac{1}{2}\|w\|^2$ que es lo mismo.

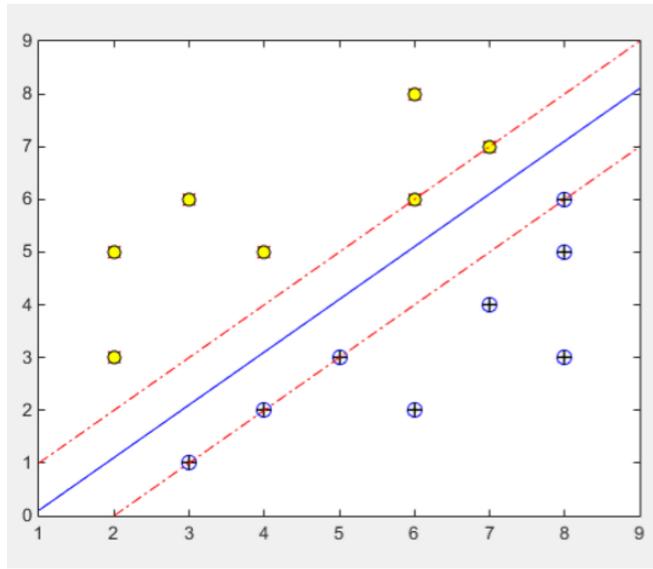


Figure 13.15: Un ejemplo de SVM

A esta altura tienen que estar convencidos que el problema de SVM se reduce a minimizar la norma del vector $\|w\|$ sujeto a las restricciones de que los puntos positivos den valores de $xw + b$ iguales a 1 y los puntos negativos den -1.

Vamos a volver a nuestro planteo entonces:

$$\begin{aligned} \min_{w,b} \frac{1}{2} \|w\|^2 \\ \text{s.t.: } y_i(wx_i + b) \geq 1 \quad , i = 1..m \end{aligned} \quad (13.52)$$

$$\begin{aligned} \min_{w,b} \frac{1}{2} \|w\|^2 \\ st : y_i(wx_i + b) - 1 \geq 0 \quad , i = 1..m \end{aligned} \tag{13.53}$$

El objetivo es desarrollar un algoritmo que nos permita resolver este problema. Podemos convertir un problema de optimización con restricciones en un problema de optimización sin restricciones usando el método de los multiplicadores de Lagrange.

13.6.4 El Problema Dual

Calculamos entonces el Lagrangiano de nuestro planteo:

$$\begin{aligned} \mathcal{L}(w, b, \alpha) &= \frac{1}{2} \|w\|^2 - \sum_{i=1}^m \alpha_i [y_i(wx_i + b) - 1] \\ \alpha_i &\geq 0 \end{aligned} \tag{13.54}$$

Una observación extremadamente importante es que minimizar el Lagrangiano respecto de w, b es equivalente a maximizarlo respecto de los α_i . Veamos el motivo: Nuestro Lagrangiano puede generalizarse de la siguiente forma:

$$\begin{aligned} \mathcal{L}(w, b, \alpha) &= f(w) - \sum \alpha_i g_i(w) \\ g_i(w) &\geq 0 \end{aligned} \tag{13.55}$$

Donde $g_i(w)$ son las restricciones $y_i(wx_i + b) - 1$

Si maximizamos los α_i entonces para se cumpla la restricción tenemos que $\mathcal{L}(w, b, \alpha) = f(w)$, ya que de no cumplirse las restricciones el Lagrangiano daría infinito. Si las restricciones se verifican entonces el valor máximo que puede tomar nuestro Lagrangiano es $f(w)$ porque el otro término necesariamente es negativo ya que tanto α_i como $g_i(w)$ tienen que ser positivos para cumplir las restricciones.

Entonces tenemos dos problemas posibles:

- Problema primal: Minimizar el Lagrangiano respecto de w, b
- Problema dual: Maximizar el Lagrangiano respecto de α

Esto es terriblemente importante!

Consideremos ahora el problema primal y queremos minimizar el Lagrangiano respecto de w, b fijando valores de α esto lo podemos lograr derivando el Lagrangiano respecto de w y b .

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= w - \sum_{i=1}^m \alpha_i y_i x_i \\ \frac{\partial \mathcal{L}}{\partial b} &= \sum_{i=1}^m \alpha_i y_i\end{aligned}\tag{13.56}$$

Igualando a cero para minimizar obtenemos algunas fórmulas interesantes:

$$w = \sum_{i=1}^m \alpha_i y_i x_i\tag{13.57}$$

$$\sum_{i=1}^m \alpha_i y_i = 0\tag{13.58}$$

Reemplazando la primera de estas dos fórmulas en el Lagrangiano y simplificando:

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m y_i y_j \alpha_i \alpha_j x_i^{(t)} x_j - b \sum_{i=1}^m \alpha_i y_i\tag{13.59}$$

Hemos visto en (11.57) que el último término se nula. Reemplazando y usando la notación de producto interno:

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m y_i y_j \alpha_i \alpha_j \langle x_i, x_j \rangle\tag{13.60}$$

Notar que de nuestro Lagrangiano han desaparecido w y b es decir que solo dependemos de los α_i y de los datos del problema x_i, y_i que son constantes.

Convirtiendo al problema dual, es decir maximizando los tenemos el siguiente planteo:

$$\begin{aligned}
 \max_{\alpha} w(\alpha) &= \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m y_i y_j \alpha_i \alpha_j \langle x_i, x_j \rangle \\
 st: \sum_{i=1}^m \alpha_i y_i &= 0 \\
 \alpha_i &\geq 0
 \end{aligned} \tag{13.61}$$

Y este es el problema para el cual vamos a construir nuestro algoritmo SVM.

Hay algo súper súper importante a notar en nuestro planteo que es que de nuestra matriz de m puntos en n dimensiones solo nos interesa el producto interno entre las filas. Si sabemos el resultado del producto interno no necesitamos conocer los puntos! Esto nos va a llevar a cosas sumamente interesantes más adelante.

Notemos que una vez que resolvamos este problema obteniendo los valores de i podemos calcular w con la ecuación (11.57). Una vez calculado w es sencillo despejar b tomando cualquier punto ya que sabemos que si el punto es positivo y está en el margen, es decir que es un support vector, entonces $wx_i^+ + b = 1 \rightarrow b = 1 - wx_i^+$. Es decir que encontrando los valores de i que maximizan (11.61) resolvemos todo el problema.

Como hemos visto SVM depende únicamente de los support vectors por lo que para aquellos puntos que no son SV el α_i correspondiente va a ser igual a cero. Es decir que la mayoría de nuestros α_i van a ser cero y aquellos que son diferentes a cero nos definen los support vectors de nuestro problema.

La cantidad de support vectors nos da una idea sobre el grado de overfitting de SVM. Cuando tenemos pocos support vectors lo que estamos diciendo es que todos los puntos que no son SV no hacen falta para clasificar a los puntos del set de entrenamiento, es decir que el algoritmo ha aprendido a generalizar muy bien. Por lo tanto la cantidad de SVs sobre el total de puntos es una métrica interesante para evaluar el grado de generalización que pudo hacer nuestro SVM. Cuanto mas chico este número mejor.

13.6.5 Algoritmo SMO (Sequential Minimal Optimization) [opcional]

Este es un algoritmo muy simple para resolver SVM basado en Coordinate Ascent, la idea básica es ir optimizando de a un α_i por vez. Fijamos todos los $\alpha_{j \neq i}$ y optimizamos α_i . Repetimos el proceso y eventualmente llegamos a los α_i que maximizan nuestro planteo.

Donde optimizar α_i implica encontrar el valor máximo de α_i i de forma tal que nuestro objetivo $W(\alpha)$ no disminuya respecto del valor anterior. En otras

Algorithm 24: SMO

```

1 while not convergence do
2   for  $i=1:m$  do
3     optimizar  $\alpha_i$ 

```

palabras mientras aumente W aumentamos α_i .

Lamentablemente tenemos un pequeño problema en nuestra idea que es la restricción

$$\sum_{i=1}^m \alpha_i y_i = 0$$

Supongamos que queremos optimizar α_1 es decir que fijamos $\alpha_2 \dots \alpha_m$

$$\begin{aligned} \alpha_1 y_1 + \sum_{i=2}^m \alpha_i y_i &= 0 \\ \alpha_1 y_1 &= -\sum_{i=2}^m \alpha_i y_i \end{aligned}$$

Multiplicando a ambos lados por y_1 (que no puede ser 0):

$$\alpha_1 y_1 y_1 = -y_1 \sum_{i=2}^m \alpha_i y_i = 0$$

Notemos que y_i^2 es 1 ya que y_i solo puede ser 1 o -1 por lo tanto:

$$\alpha_1 = -y_1 \sum_{i=2}^m \alpha_i y_i = 0$$

Es decir que el valor de α_1 depende de los demás alfas y no podemos moverlo en ninguna dirección.

La solución es que tenemos que optimizar los alfas de a pares!.

Algorithm 25: SMO

```

1 while not convergence do
2   Select a pair  $\alpha_i, \alpha_j$ 
3   optimizar

```

La selección del par de variables a optimizar puede realizarse con alguna heurística. O simplemente al azar.

Para la fase de optimización necesitamos pensar cómo podemos variar dos variables al mismo tiempo de forma tal de maximizar el objetivo.

Supongamos que queremos optimizar α_1 y α_2 , sabemos que:

$$\alpha_1 y_1 + \alpha_2 y_2 = - \sum_{i=3}^m \alpha_i y_i$$

La parte de la derecha de la ecuación es constante porque son los valores de que quedan fijos, lo reemplazamos por ζ (zeta)

$$\alpha_1 y_1 + \alpha_2 y_2 = \zeta$$

$$\alpha_1 = (\zeta - \alpha_2 y_2) y_1$$

Y ahora podemos reemplazar el valor de α_1 en w :

$$w(\alpha_1, \alpha_2, \dots, \alpha_m) = w((\zeta - \alpha_2 y_2) y_1, \alpha_2, \dots, \alpha_m)$$

Usando la ecuación de (11.57) es fácil darnos cuenta que W queda como una ecuación cuadrática en función de α_2 ya que todos los demás valores son constantes. Podemos resolver entonces α_2 simplemente derivando w e igualando a cero. De esta forma podemos resolver todo el problema planteado por SVM con una algoritmo extremadamente sencillo:

Algorithm 26: SMO

Data: X,y
Result: w,b, α

```

1 while not convergence do
2   Select a pair  $\alpha_i, \alpha_j$ 
3   despejar  $\alpha_j$  en función de  $\alpha_i$ 
4   reemplazar en  $w(\alpha)$ 
5   resolver  $\alpha_j$ 
6   resolver  $\alpha_i$ 
7 resolver  $w, b$ 

```

Este algoritmo muy simple funciona en todos los casos en los cuales los datos sean linealmente separables, si los datos no son linealmente separables el algoritmo falla, esto sería malo, peor que Perceptrón o Logistic Regression así que vamos a introducir un pequeño cambio en nuestro algoritmo para permitir que funcione cuando los datos no son linealmente separables.

13.6.6 Soft Margin

Recordando, allá lejos y hace tiempo cuando vimos Perceptrón vimos que cuando los datos no eran linealmente separables Perceptrón intentaba minimizar la cantidad de datos mal clasificados. En la simpleza de Perceptrón cualquier hiperplano con igual cantidad de puntos mal clasificados era igual en cuanto a la solución del problema.

Ocurre que no sólo es bueno clasificar bien los puntos sino que a veces puede ser bueno clasificar mal algunos puntos. Veamos el siguiente caso:

Tenemos claramente dos grupos, uno de cruces y otro de circulitos con un punto alejado que aparentemente está mal clasificado. Puede ser un error de medición, un caso excepcional o cualquier otra cosa, a estos puntos los llamamos “outliers” y suelen ser todo un problema por lo que más adelante veremos un algoritmo para detectarlos y filtrarlos. La línea sólida es una separación lineal perfecta tal y como la encontraría Perceptrón ya que tiene cero errores de clasificación. La línea punteada en cambio es la separación que nos gustaría encontrar ya que es mucho más efectiva para clasificar puntos a futuro. Suponemos que esto queda completamente claro.

La idea entonces es introducir en SVM una penalización por cada punto mal clasificado. Vamos a escribir entonces:

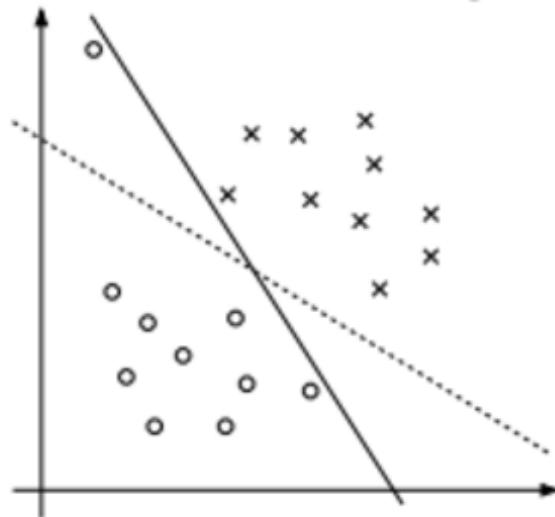


Figure 13.16: Soft Margin

Tenemos, claramente, dos grupos: uno de cruces y otro de circulitos con un punto alejado que aparentemente está mal clasificado. Puede ser un error de medición, un caso excepcional o cualquier otra cosa, a estos puntos los llamamos “outliers” y suelen ser todo un problema por lo que más adelante veremos un algoritmo para detectarlos y filtrarlos. La línea sólida es una separación lineal perfecta tal y como la encontraría Perceptrón ya que tiene cero errores de clasificación. La línea punteada en cambio es la separación que nos gustaría encontrar ya que es mucho más efectiva para clasificar puntos a futuro. Suponemos que esto queda completamente claro.

La idea entonces es introducir en SVM una penalización por cada punto mal

clasificado. Vamos a escribir entonces:

$$y_i(wx_i + b) \geq 1 - \xi_i \quad (13.62)$$

Donde “xi” es el error que tenemos para el punto “i” es decir que está dentro del margen. Tiene margen funcional menor a 1. ξ_i va a ser cero para todos los puntos por afuera del margen y un cierto valor positivo para los puntos dentro del margen. Agreguemos ahora un costo constante “C” para penalizar estos errores:

El planteo entonces es:

$$\begin{aligned} & \min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^m \xi_i \\ & st : y_i(wx_i + b) \geq 1 - \xi_i \\ & \quad \xi_i \geq 0 \end{aligned} \quad (13.63)$$

“C” regula si el algoritmo va a intentar minimizar la cantidad de puntos dentro del margen o bien maximizar el margen entre las clases. Con un margen muy amplio tendremos muchos puntos dentro del mismo y con un margen muy pequeño muy pocos, en general queremos encontrar un equilibrio.

Aplicando el Lagrangiano y convirtiendo el problema al dual tenemos:

$$\begin{aligned} \max_{\alpha} w(\alpha) = & \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m y_i y_j \alpha_i \alpha_j < x_i, x_j > \\ & st : \sum_{i=1}^m \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \end{aligned} \quad (13.64)$$

Notemos que agregando el concepto de soft-margin el único cambio que tenemos en nuestro planteo es que los α que antes estaban solamente restringidos a ser positivos ahora tienen que ser también menores o iguales al factor de penalización C.

Veamos cómo influye esto en nuestro algoritmo SMO:

13.6.7 SMO (Soft Marging)

Recordemos que seleccionando α_1 y α_2 para optimizar teníamos:

$$\alpha_1 y_1 + \alpha_2 y_2 = \zeta$$

Y ahora cada α_i debe ser menor o igual a "C" es decir que graficamente tenemos lo siguiente:

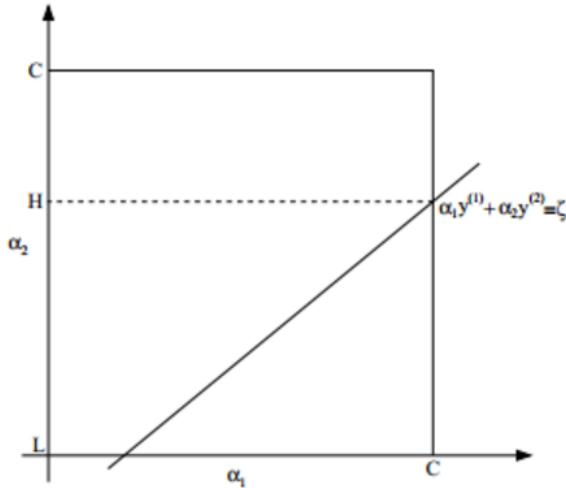


Figure 13.17: SMO Soft Margin

Como vemos los α_i tienen que estar dentro de $[0,C] \times [0,C]$. Y al mismo tiempo tienen que satisfacer $\alpha_1 y_1 + \alpha_2 y_2 = \zeta$. Es decir que tenemos que los valores válidos para α_1 y α_2 son aquellos que están al mismo tiempo dentro del cuadrado y sobre la línea. Es decir que α_2 que es el valor que despejamos no puede ser menor a "L" ni mayor a "H". En nuestro ejemplo $L=0$ pero podría ser cualquier otro valor según la recta $\alpha_1 y_1 + \alpha_2 y_2 = \zeta$.

Podemos dejar nuestro algoritmo sin cambios pero una vez que encontramos el valor de α_2 que maximiza w realizamos el siguiente ajuste:

Si

$$\alpha_2 < L \rightarrow \alpha_2 = L$$

Si

$$\alpha_2 > H \rightarrow \alpha_2 = H$$

En este momento declaramos el planteo completo y tenemos un algoritmo listo para programar que nos permite optimizar SVM usando Soft-Margin.

Hemos visto la versión lineal de SVM pero SVM puede resolver también separaciones no-lineales mediante un truco muy bonito que llamamos "the kernel trick".

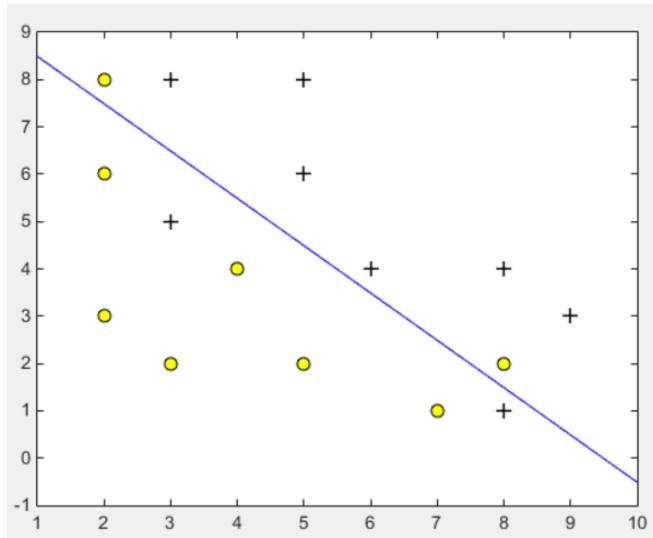


Figure 13.18: Un ejemplo de SVM con Soft Margin

13.7 The Kernel Trick

“The Kernel Trick” es un mecanismo muy ingenioso que nos permite clasificar datos que no son linealmente separables con un clasificador lineal. El truco está basado en el teorema de Cover:

Theorem 27 (Cover) *Dado un set de datos que no es linealmente separable es posible, con una alta probabilidad transformarlo en un set de datos linealmente separable proyectándolo en un espacio de mas dimensiones mediante alguna transformación no-lineal*

La demostración de este teorema es muy sencilla: dados n puntos podemos asignar a cada punto un vértice del simplice de $n - 1$ dimensiones y entonces cualquier partición de los puntos en dos conjuntos es separable por un hiperplano.

A modo de ejemplo consideremos 4 puntos en el plano, hay casos en los cuales estos 4 puntos no son linealmente separables, el teorema de Radon dice que en n dimensiones existe un conjunto de $n + 2$ puntos tales que sus fronteras convexas se intersecten, cuando las fronteras convexas se intersectan los conjuntos no son linealmente separables. En el plano el caso es el XOR en donde 4 puntos divididos en dos conjuntos $s1 = (0, 0), (1, 1)$ y $s2 = (1, 0), (0, 1)$ no son linealmente separables. Por el teorema de Cover podemos proyectar estos 4 puntos en $n - 1 = 3$ dimensiones. Si ponemos cada punto en la esquina de un cubo en 3d cualquier partición de esos 4 puntos en dos conjuntos es linealmente separable.

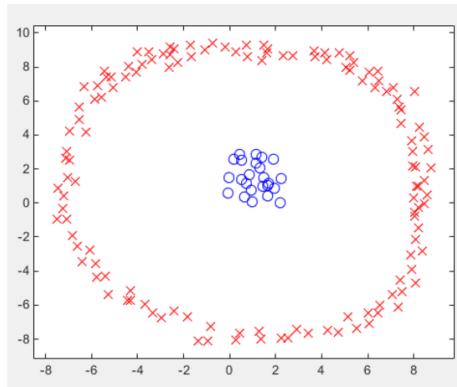


Figure 13.19: Puntos que no son linealmente separables en el plano

La figura 13.19 muestra un conjunto de puntos que no podemos separar linealmente en el plano. El resultado de un clasificador lineal es en general bastante malo.

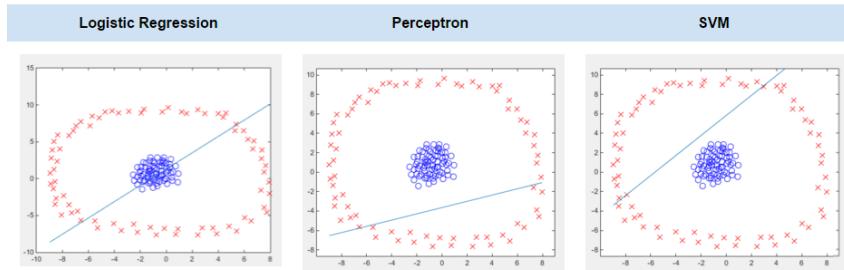


Figure 13.20: Clasificadores lineales con clases que no son linealmente separables.

Tanto SVM como Perceptrón buscan clasificar bien a todos los puntos azules y una cierta cantidad de puntos rojos, Logistic Regression por otra parte divide ambas clases en dos partes y clasifica mal a la mitad de los puntos. Las dos interpretaciones del problema están tan mal y bien como uno quiera interpretarlas. En definitiva lo ideal sería poder separar las clases pero es evidente que esto es imposible con un clasificador lineal. O eso parece...

El problema que tenemos es que la cantidad de dimensiones en las que estamos trabajando no son suficientes para separar los datos, si pasamos los datos a tres dimensiones podemos ponerles una cierta altura en el eje "z" a los puntos azules, por ejemplo 1 y dejar los puntos rojos en z=0.

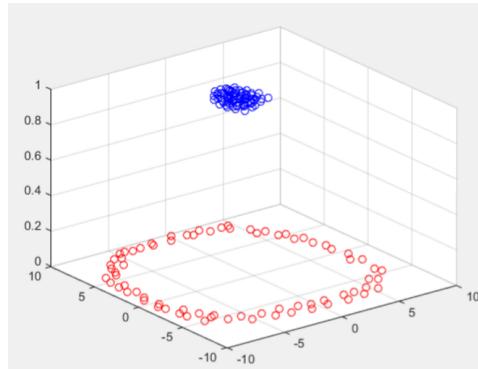


Figure 13.21: Agregando una dimensión

En este espacio es muy simple separar los puntos con un simple hiperplano, es decir con un clasificador lineal como cualquiera de los que hemos usado hasta el momento. Segundo parece lo único que necesitamos entonces es mapear los puntos de dos dimensiones a tres dimensiones y el problema está resuelto.

La función de mapeo la vamos a llamar $\phi(x)$ que recibe un punto x en n dimensiones y devuelve otro punto x en una cantidad de dimensiones mayor.

Ahora debemos notar que en SVM en ningún momento necesitamos acceder a los puntos de nuestro conjunto de datos sino simplemente al producto interno entre los mismos. Por eso en SMO SVM cuando los datos son pocos pre-calculamos la matriz K de productos internos. Si agregamos la función de mapeo al producto interno tenemos lo que llamamos un “kernel”

$$K(x, y) = \langle \phi(x), \phi(y) \rangle \quad (13.65)$$

El “kernel trick” consiste en calcular el resultado de $K(x, y)$ sin realizar el mapeo de los datos, es decir directamente a partir de x e y calcular cuál será el resultado de su producto interno luego de una cierta transformación que lleva a los puntos a más dimensiones, repetimos, sin realizar la transformación en ningún momento.

Veamos un ejemplo:

Supongamos que tenemos puntos en tres dimensiones: $x = (x_1, x_2, x_3); y = (y_1, y_2, y_3)$. Definimos la siguiente función que mapea de tres a nueve dimensiones:

$$\phi(x) = (x_1x_1, x_1x_2, x_1x_3, x_2x_1, x_2x_2, x_2x_3, x_3x_1, x_3x_2, x_3x_3)$$

Calculamos ahora el producto interno $\langle \phi(x), \phi(y) \rangle$:

$$\begin{aligned}
\langle \phi(x), \phi(y) \rangle &= x_1x_1y_1y_1 + x_1x_2y_1y_2 + x_1x_3y_1y_3 + x_2x_1y_2y_1 + x_2x_2y_2y_2 + x_2x_3y_2y_3 \\
&\quad + x_3x_1y_3y_1 + x_3x_2y_3y_2 + x_3x_3y_3y_3 \\
&= x_1^2y_1^2 + x_1x_2y_1y_2 + x_1x_3y_1y_3 + x_2x_1y_2y_1 + x_2^2y_2^2 + x_2x_3y_2y_3 \\
&\quad + x_3x_1y_3y_1 + x_3x_2y_3y_2 + x_3^2y_3^2 \\
&= x_1^2y_1^2 + x_2^2y_2^2 + x_3^2y_3^2 + 2(x_1y_1x_2y_2) + 2(x_1y_1x_3y_3) + 2(x_2y_2x_3y_3) \\
&= (x_1y_1 + x_2y_2 + x_3y_3)^2 \\
&= \langle x, y \rangle^2
\end{aligned} \tag{13.66}$$

Esto quiere decir que si tenemos por ejemplo $x = (1, 2, 2)$ $y = (3, 2, 1)$ podemos calcular $K(x, y) = \langle x, y \rangle^2 = (3+4+2)^2 = 81$ Lo cual es muchísimo más sencillo que convertir cada vector de tres a nueve dimensiones y luego realizar el producto interno entre estos vectores de nueve dimensiones. Esta es precisamente la gracia del “kernel trick” obtener de forma sencilla el resultado de un producto interno en un espacio de mayores dimensiones al original.

Veamos ahora el primero de los kernels que podemos usar con SVM, se trata del kernel polinómico:

13.7.1 Kernel Polinómico

$$K(x, y) = (\langle x, y \rangle + c)^d \tag{13.67}$$

Para entender el funcionamiento de este kernel vamos un ejemplo con “d”=2 y puntos x, y de tres dimensiones: $x = (x_1, x_2, x_3); y = (y_1, y_2, y_3)$

Haciendo cuentas que vamos a omitir por cuestiones de espacio y porque no son mas que cuadrados de trinomios llegamos a que podemos encontrar la función de mapeo asociada al kernel (ϕ)

$$\phi(x) = [x_1x_1, x_1x_2, x_1x_3, x_2x_1, x_2x_2, x_2x_3, x_3x_1, x_3x_2, x_3x_3, \sqrt{2}cx_1, \sqrt{2}cx_2, \sqrt{2}cx_3, c]$$

Es decir que calcular $(\langle x, y \rangle + c)^2$ equivale a realizar el producto interno entre x e y transformados a 13 dimensiones por $\phi(x)$.

Debemos preguntarnos que tipo de kernels son válidos ya que podemos sospechar que no cualquier cosa que hagamos con dos vectores y que devuelva un escalar es un kernel. Para que sea un kernel tiene que ser posible hallar la función tal que el resultado del kernel sea igual al producto interno entre los dos vectores mapeados por $\phi(x)$. Para poder llegar a una definición vamos a definir antes la llamada “Matriz del Kernel”.

Sea $K(x, y)$ un kernel y sea X un conjunto de “m” puntos en “n” dimensiones. definimos a la matriz K como la matriz en la cual $K_{ij} = K(x_i, x_j)$

Esta matriz es muy parecida a la matriz de productos internos que podemos pre-calcular en SVM solo que en SVM lineal $K(x, y) = \langle x, y \rangle$. Es decir que el kernel lineal es el producto interno.

13.8 Teorema de Mercer

Theorem 28 (Mercer) *Sea $K : \mathbb{R}^n \rightarrow \mathbb{R}^n$ una función. Decimos que K es un kernel válido si y solo sí para cualquier conjunto de puntos $x_1, x_2, \dots, x_m \in \mathbb{R}^n$ la matriz asociada a K es simétrica y positiva semi-definida.*

Una matriz K es positiva y semi-definida si y solo si para cualquier z se verifica que $zKz^{(t)} \geq 0$. Es condición necesaria y suficiente que la matriz no tenga autovalores negativos.

Supongamos que K es un kernel válido de forma tal que $K_{ij} = K(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$

Por propiedad del producto interno: $K_{ij} = \langle \phi(x_i), \phi(x_j) \rangle = \langle \phi(x_j), \phi(x_i) \rangle = K(x_j, x_i) = K_{ji}$ esto demuestra que K debe ser simétrica.

Desarrollemos la segunda condición del teorema usando $z^{(j)}$ para denotar la j -ésima coordenada de z

$$\begin{aligned}
zKz^{(t)} &= \sum_i \sum_j z^{(i)} K_{ij} z^{(j)} \\
&= \sum_i \sum_j z^{(i)} \langle \phi(x_i), \phi(x_j) \rangle z^{(j)} \\
&= \sum_i \sum_j z^{(i)} \sum_k \phi^{(k)}(x_i) \phi^{(k)}(x_j) z^{(j)} \\
&= \sum_k \sum_i \sum_j z^{(i)} \phi^{(k)}(x_i) \phi^{(k)}(x_j) z^{(j)} \\
&= \sum_k \left(\sum_i z^{(i)} \phi^{(k)}(x_i) \right)^2 \\
zKz^{(t)} &\geq 0
\end{aligned} \tag{13.68}$$

En el ante-último paso usamos la siguiente propiedad: $\sum_i \sum_j z^{(i)} x^{(i)} x^{(j)} z^{(j)} = \langle x, z \rangle^2$

Con lo cual queda demostrado el teorema de Mercer.

Destaquemos que el hecho de que la matriz sea simétrica y positiva semi-definida es condición necesaria y suficiente, es decir que si verificamos estas dos condiciones en algún kernel entonces sabemos que existe el mapeo $\phi(x)$ sin necesidad de calcularlo.

Al ser la matriz del Kernel una matriz simétrica y positiva podemos tomarla como una matriz de semejanza entre los puntos. En el caso del kernel que no hace nada, es decir el producto interno el mismo da cero cuando los puntos son diferentes (ortogonales) o un número cuando los puntos son iguales. Es por esto que muchas funciones de semejanza pueden usarse como Kernel al cumplir las condiciones de Mercer, un ejemplo es el Kernel Gaussiano también conocido como RBF.

13.8.1 Kernel Gaussiano (RBF)

$$K(x, y) = \exp\left(-\frac{\|x-y\|^2}{2\sigma^2}\right) \quad (13.69)$$

Sigma es un hiper-parámetro que depende de los datos a los cuales aplicamos el kernel. Esta es una función de semejanza muy usada que da como resultado algo cercano a cero cuando x e y están muy lejos y da como resultado algo cercano a 1 cuando x e y son muy similares.

Este Kernel cumple las condiciones de Mercer por lo que sabemos que es un kernel válido y que por lo tanto existe una función de mapeo , en este caso la función mapea a los vectores x e y a un espacio de infinitas dimensiones(!), no podemos calcular $\phi(x)$ pero sabemos que existe por el teorema de Mercer que lo garantiza.

Esto es llevar el “kernel trick” al extremo, mediante una operación sencilla estamos calculando el equivalente al producto interno de dos vectores de infinitas dimensiones.

Tomando nuestro conjunto de datos “inseparables” podemos aplicar el kernel RBF y usar el resultado como una tercera dimensión para visualizar si sirve para separar los datos:

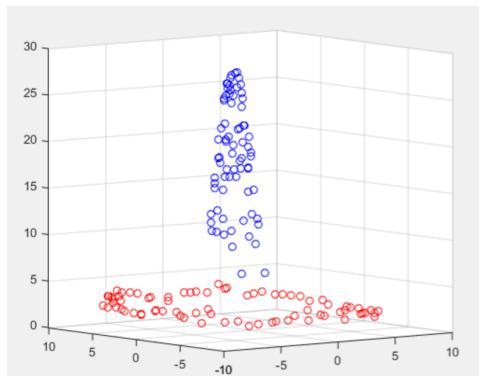


Figure 13.22: Aplicando el Kernel RBF

Sorpresa! Ahora cualquier clasificador lineal va a estar en condiciones de separar nuestros datos en dos clases sin problema alguno.

Vamos a analizar el efecto del kernel en las distancias entre los puntos

Este es el resultado con los puntos como referencia.

Los puntos 1 a 75 son los azules. El sector superior izquierdo de la matriz muestra las distancias entre los mismos que son muy chicas (oscuras). El sector inferior derecho muestra las distancias entre los puntos rojos, como vemos algunos están muy cercanos entre sí por ser vecinos y otros están en extremos opuestos del anillo rojo y tienen distancia muy pequeña. Los otros dos cuadrantes representan las distancias entre puntos azules y rojos que son bastante similares.

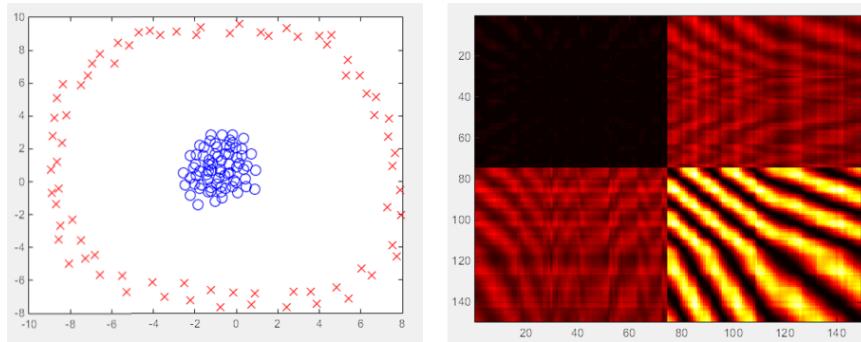


Figure 13.23: Distancias sin aplicar el Kernel

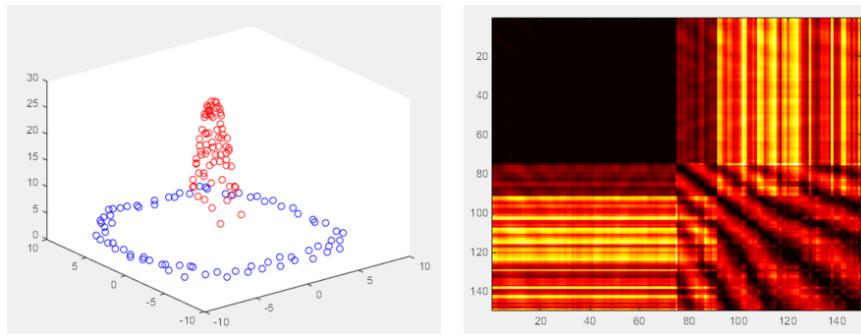


Figure 13.24: Distancias luego de aplicar RBF

Aquí vemos el efecto de la proyección del set de datos en 3d con respecto a las distancias. Nos interesan especialmente los sectores que representan las distancias entre los puntos rojos y los azules, ahora ya no es todo equidistante lo cual nos ayuda a pensar que tenemos mejores posibilidades de encontrar una separación lineal entre las clases. Notar que el Kernel mantiene la cercanía entre los puntos azules y los rojos.

13.9 Aproximación de Nystrom

Uno de los problemas al usar kernels es la necesidad de calcular la matriz del kernel que si los puntos son muchos puede ser potencialmente enorme. Cuando el cálculo de la matriz completa no es factible aparece la aproximación de Nystrom como una solución. Los pasos son los siguientes:

1. Elegir al azar una cierta cantidad de puntos de nuestro conjunto de datos.

2. Calcular la matriz del Kernel para estos puntos (siendo la cantidad un número manejable)

Cuando queremos calcular $K(x, y)$ si los puntos están en la matriz entonces el resultado es automático. Si los puntos no están en la matriz entonces hay que calcular los puntos de la matriz más cercanos a x , los puntos más cercanos a y y finalmente estimar $K(x, y)$ como el promedio ponderado de los $K(w, z)$ en la matriz en donde w es cercano a x y z es cercano a y .

Como podemos imaginar cuantos más puntos tomemos para la matriz mejor será la aproximación. Existen muchas variantes de este algoritmo de aproximación y varios de los métodos que usan kernels usan alguna de estas variantes. Recordemos que la matriz no es necesaria, podemos calcular $K(x, y)$ cada vez que sea necesario. Cuando la memoria alcanza y queremos velocidad pre-calculamos la matriz. Si la matriz no entra en memoria y necesitamos performance usaremos la aproximación de Nystrom, finalmente si la matriz no entra en memoria y no tenemos problemas de performance entonces simplemente calculamos el valor de $K(x, y)$ cada vez que sea necesario.

El “kernel trick” es un recurso realmente sorprendente que permite que un clasificador lineal separe datos que no son linealmente separables.

13.10 Kernel SVM

En esta sección aplicamos el Kernel trick al algoritmo SVM. Esto es trivial ya que en toda la formulación de SVM teníamos todo expresado en función de productos internos por lo que solo tenemos que reemplazar esos productos internos por el valor de $K(x, y)$ y el algoritmo funciona perfectamente.

El objetivo de esta sección es hablar sobre los hiper-parámetros de SVM usando un kernel RBF, tenemos dos hiper-parámetros a optimizar: “C” que es el costo de clasificar mal usando soft-margin y σ que es el radio de la función RBF para el kernel gaussiano.

El ejemplo muestra como un SVM con Kernel RBF es capaz de encontrar una frontera no-lineal para separar las clases. En líneas generales este algoritmo puede encontrar prácticamente cualquier tipo de frontera, igual que KNN pero con la ventaja de solo usar los puntos cercanos a la frontera para clasificar, es decir los support vectors.

13.10.1 El efecto de C y σ

El parámetro “C” de SVM y los parámetros que necesita el Kernel seleccionado son críticos para el funcionamiento de SVM. Vamos a ver en primer lugar un ejemplo de diferentes soluciones de SVM con diferentes hiper-parámetros, luego vamos a analizar como influye cada uno.

Comenzamos por un ejemplo de soluciones que SVM encuentra para un cierto set de datos y diferentes hiper-parámetros:

“C” controla la penalización por clasificar mal, a medida que aumentamos “C” SVM va a intentar clasificar mayor cantidad de puntos bien en el set de

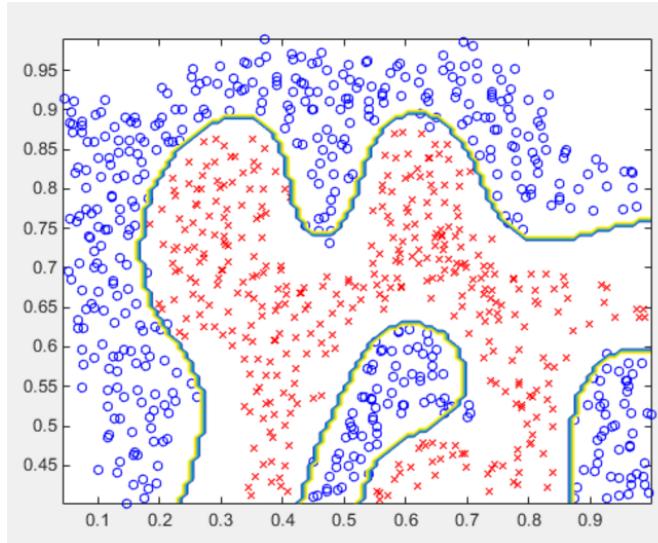


Figure 13.25: Ejemplo de frontera usando Kernel RBF

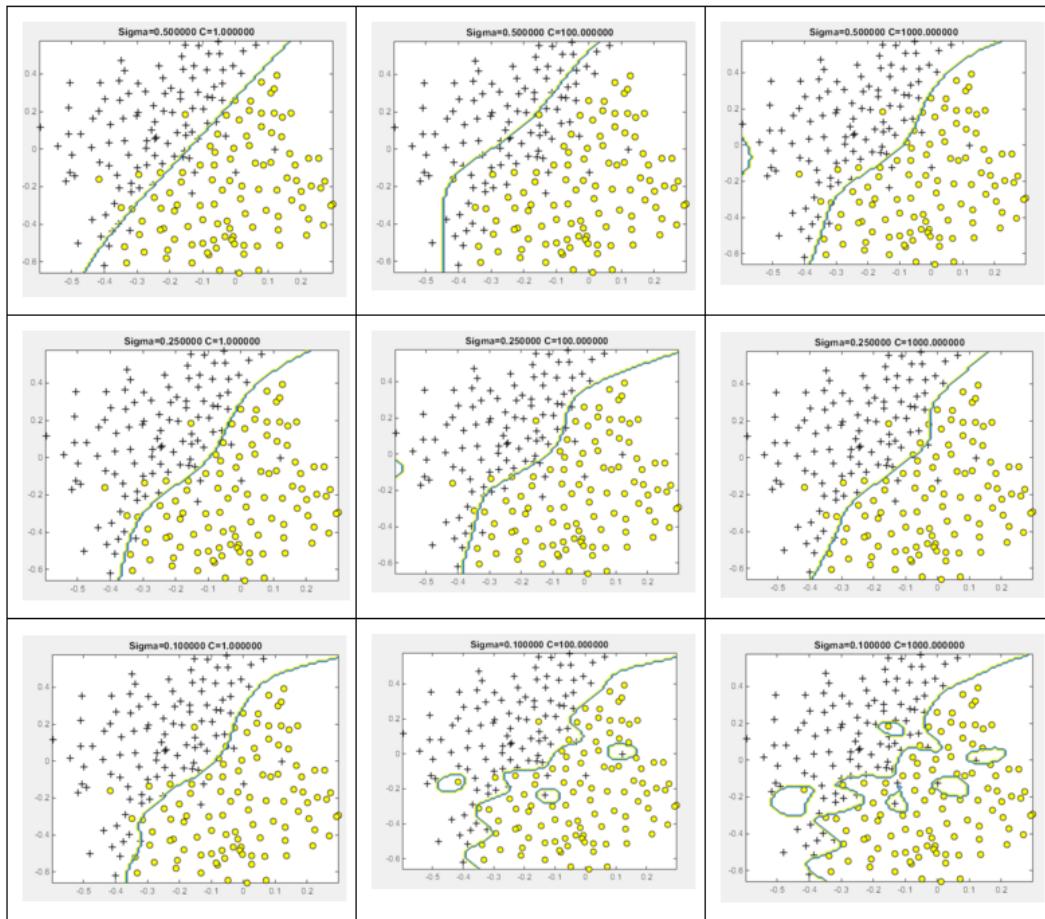
entrenamiento. Un “C” muy chico hace que el modelo no responda adecuadamente a los datos (high bias) mientras que un “C” muy grande hace que el modelo sobre-ajuste (high variance) es decir que cae en overfitting.

Sigma por otro lado controla la zona alrededor de cada punto o en otras palabras cuando dos puntos son considerados cercanos entre sí. Un sigma muy chico hace que los puntos no tengan vecinos y queden aislados unos de otros esto ocasiona high variance. Un sigma muy grande hace que todos los puntos sean vecinos lo cuál hace que el modelo no ajuste bien a los datos (high bias).

En nuestro caso tenemos el modelo con mayor bias y menor varianza en la esquina superior izquierda ($C=1$. $\Sigma=0.5$). Como vemos la solución es similar a una recta y esto refleja que el modelo no está ajustando bien a los puntos. En el otro extremo tenemos $C=1000$ y $\Sigma=0.1$ y vemos un claro caso de overfitting (sobre-ajuste) ya que prácticamente tenemos una clasificación por cada punto, es decir que tendremos muy buen ajuste al set de datos pero muy mala generalización.

En general SVM es un algoritmo que resiste muy bien el overfitting, o dicho de otra forma es bastante difícil caer en overfitting usando un SVM, especialmente si tenemos muchos datos. Cuando la cantidad de datos es grande SVM va a intentar siempre la menor cantidad de support vectors posible, al hacer esto automáticamente el algoritmo está intentando generalizar bien ya que vamos a clasificar muchos puntos usando una cantidad relativamente pequeña de de puntos-frontera.

Al igual que en todos los algoritmos los hiper-parámetros óptimos para SVM se buscan mediante grid-search. Una opción recomendable es realizar una matriz

Figure 13.26: El efecto de C y σ

Sigma chico = High Variance, Low Bias	Sigma grande = Low Variance, High Bias
C chico = Low Variance, High Bias	C grande = High Variance, Low Bias

Table 13.6: Bias y Variance en SVMs.

con el error para el set de validación en función de los parámetros C y σ que vamos probando, a medida que vamos encontrando "zonas" en donde el algoritmo funciona mejor podemos ir refinando los valores para estos hiper-parámetros. Siempre es conveniente también controlar la cantidad de support-vectors que el algoritmo usa, una cantidad excesivamente grande es sinónimo de un funcionamiento anómalo en SVM y debe ser corregido.

13.11 SVM Transductivo

SVM al igual que todos los clasificadores que hemos visto es un método *inductivo* es decir que aprende a partir de los datos del set de entrenamiento que, suponemos, se encuentran bien clasificados. Sin embargo los datos no clasificados son muchos mas que los que ya tienen un label. El uso de estos puntos para los cuales no conocemos su clase se conoce como *transducción*. Este es un caso de aprendizaje semi-supervisado ya que usamos puntos con labels y puntos sin label al mismo tiempo.

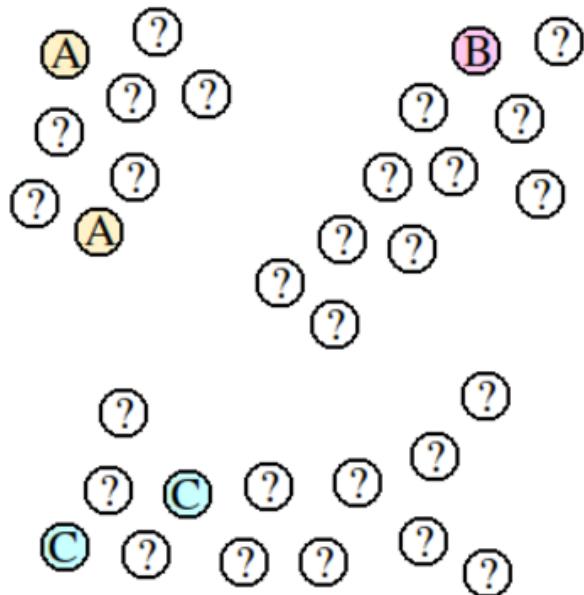


Figure 13.27: SVM Transductivo

En la figura 13.27 vemos que si solo usamos A,B y C como set de entrenamiento podríamos llegar a un modelo equivocado. Si agregamos puntos cuya clase desconocemos el modelo puede mejorar aprendiendo mejor la verdadera frontera entre las clases, los puntos sin clasificar siempre son realmente muchos por lo que ayudan a definir fronteras mas precisas. Por supuesto que el detalle es que no conocemos a que clase pertenecen estos puntos.

La formulación de SVM con transducción es la siguiente:

$$\begin{aligned}
 & \min \frac{1}{2} \|w\|^2 + C \sum \xi \\
 & sty_i(wx_i - b) \geq 1 \\
 & y_j^*(wx_j - b) \geq 1 \\
 & y_j^* \in \{-1, 1\}
 \end{aligned} \tag{13.70}$$

Como vemos agregamos una restricción en donde el SVM puede darle el label que quiera a los puntos que no tienen label pero cumpliendo el objetivo de maximizar el margen de separación de las clases para los puntos para los cuáles conocemos su label.

Esto agrega bastante complejidad a SVM y será bastante mas lento pero el algoritmo es igual al que hemos desarrollado. Para muchos autores el aprendizaje semi-supervisado es el futuro de la clasificación automática y los SVM transductivos son una de las opciones mas interesantes a explorar en el futuro cercano.

13.12 SVM Ordinal

Podemos usar SVM para aprender a ordenar, cada documento tendrá asociada una clase C_i de forma tal que si un documento tiene una clase mayor que otro entonces es mas importante. Esto no es igual a un problema de clasificación multiclas sino que es un problema de *clasificación ordinal*

$$C_i > C_j \rightarrow wx_i > wx_j \rightarrow w(xi - xj) > 0 \tag{13.71}$$

Podemos ahora usar $x_i - x_j$ como puntos de nuestro set de entrenamiento y +1 o -1 como clase y entrenar un SVM común.

Este tipo de SVM es ideal para problemas de rankeo de tipo Learning to Rank, que hemos visto en el capítulo sobre Information Retrieval y que vamos a volver a ver dentro del marco de los sistemas de recomendaciones.

13.13 Conclusiones

Desde que fue desarrollado por Vladimir Vapnik en la década del 90 los SVMs han ganado mas y mas popularidad, principalmente por haber sido durante un tiempo el estado del arte para la clasificación de textos e imágenes. Cuando los datos tienen muchas dimensiones un SVM lineal puede encontrar una separación pero al mismo tiempo puede decidir cuáles son los puntos críticos para determinar la frontera entre las clases (support vectors). Cuando las clases

no son linealmente separables un SVM con kernel RBF suele dar muy buenos resultados.

SVM es un pariente cercano de KNN porque se basa en la semejanza (kernel) entre los puntos para determinar la frontera entre las clases, pero el comportamiento de ambos algoritmos es totalmente diferente. KNN tiene como problema la maldición de la dimensionalidad mientras que SVM brilla cuando podemos proyectar los datos en infinitas dimensiones para encontrar una separación lineal. Es uno de los pocos algoritmos que navega por hiper-espacios de enorme dimensionalidad de forma confiable.

Chapter 14

Clustering

Figures don't lie but liars do figure.

- Mark Twain

El proceso que conocemos como *clustering* es el ejemplo mas claro de aprendizaje no-supervisado. Dado un set de datos queremos agrupar los datos en *clusters* de forma tal que todos los datos dentro de un mismo cluster sean similares entre sí pero diferentes a los de cualquier otro cluster. Algunos estudios sugieren que esta es una forma de aprendizaje universal y que es la forma principal de aprendizaje del cerebro humano, mediante la asociación de objetos aprendemos su significado de la misma forma que un niño es capaz de determinar que cosas se pueden comer y que cosas sirven para jugar aun al verlas por primera vez.

El aprendizaje no-supervisado no necesita saber a que clase pertenece cada objeto, es decir que solo necesita los objetos mismos. En el mundo real la gran mayoría de los datos que podemos recolectar son de este tipo, no tienen *labels*.

Los algoritmos de clustering son importantes en todo tipo de aplicaciones, por ejemplo un supermercado o tienda online puede agrupar productos que se suelen vender juntos o enviarnos recomendaciones de productos similares a los que compramos. Podemos agrupar música y luego encontrar temas similares a los que nos gustan analizando de que clusters provienen esos temas. Podemos reconocer imágenes u objetos, etc.

Para algunos autores el aprendizaje no-supervisado es el área que mas va a necesitarse en el futuro, estos algoritmos son los que mas datos van a tener en el futuro por poder trabajar sin necesidad de labels, si los datos son el nuevo petróleo estos algoritmos son las refinerías. Los algoritmos de clustering están agazapados, esperando que llegue su gran momento para dominar el mundo.

14.1 Clustering Jerárquico

El método de clustering jerárquico, también llamado aglomerativo es uno de los algoritmos de clustering más importantes ya que genera una descomposición jerárquica del set de datos creando clusters desde $k = 1$ hasta m es decir la cantidad de puntos de nuestro set de datos. Es útil tanto como herramienta de clustering como así también como instrumento para el análisis exploratorio de los datos ya que tanto la descomposición creada como su representación gráfica aportan información que ayuda en gran medida a entender los datos. El gran problema del clustering jerárquico es su eficiencia para grandes volúmenes de datos, tema del cual nos ocuparemos más adelante.

El algoritmo en sí es extremadamente sencillo, comenzamos suponiendo que cada punto es un cluster es decir que tenemos m clusters inicialmente. Luego en cada paso tomamos los dos clusters más cercanos entre sí y los unimos en un nuevo cluster. Este proceso se repite hasta que tenemos un único cluster. El paralelo con los códigos de Huffman es evidente.

Hay dos cuestiones importantes que tenemos que definir para poner en marcha el algoritmo:

- Que distancia usar
- Como calcular la distancia entre dos clusters

Sobre las distancias ya hemos hablado a lo largo de varios capítulos, al igual que en KNN tenemos que elegir la métrica que mejor se ajuste a nuestro set de datos.

14.1.1 Distancia Entre Clusters

En el primer paso comparamos los datos todos contra todos y elegimos los dos objetos mas similares. Tenemos ahora $m - 1$ clusters, 1 cluster de tamaño 2 y $m - 1$ clusters de tamaño 1. En el paso siguiente hay que volver a elegir los dos clusters mas cercanos pero ahora no alcanza con comparar item a item porque uno de nuestros clusters tiene mas de un ítem. Necesitamos una forma de medir la distancia entre dos *grupos* de ítems.

Las opciones son:

- La distancia mínima entre dos puntos de cada cluster
- La distancia máxima entre dos puntos en cada cluster
- El promedio de las distancias de todos los puntos de un cluster contra todos los puntos del otro cluster
- La distancia entre los promedios de los puntos de cada cluster (centroides)
- El método de Ward [Ward]

14.1.2 Encontrando la Cantidad de Clusters

Veamos ahora de que forma podemos usar clustering jerárquico para determinar la cantidad de clusters que hay en nuestro set de datos. Vamos a usar el siguiente ejemplo en el plano con la distancia euclideana.

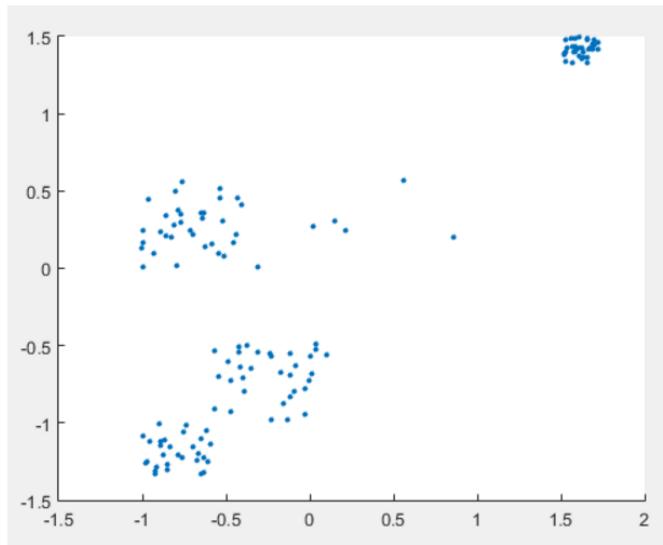


Figure 14.1: Puntos para Clustering Jerárquico

Aplicamos el algoritmo completo, es decir hasta que nos queda un único cluster con todos los puntos. El resultado es una lista de que clusters debemos unir en cada paso, por ejemplo si en el primer paso unimos el punto 100 y el 121 la primera fila de nuestra lista es (100,121) y a partir de ahora existe el cluster $m + 1$ que representa a estos dos puntos.

Ejemplo: Si tenemos 5 puntos y la siguiente lista:

```
1 3
6 5
2 4
7 8
```

Unimos el punto 1 y el 3 en el cluster 6 (los puntos son 5) el cluster 6 es (1,3). Luego unimos el cluster 6 y el 5 en el cluster 7 (1,3,5) luego el cluster 2 y 4 en el cluster 8 (2,4) finalmente el cluster 7 y el 8 (1,3,5,2,4).

En cada paso podemos indicar la distancia entre los clusters que sabemos es la mínima de todas las posibles distancias entre clusters. Por ejemplo:

```
1 3 2
6 5 2.5
```

2	4	3
7	8	12

Podemos graficar la distancia para nuestro set de 135 puntos:

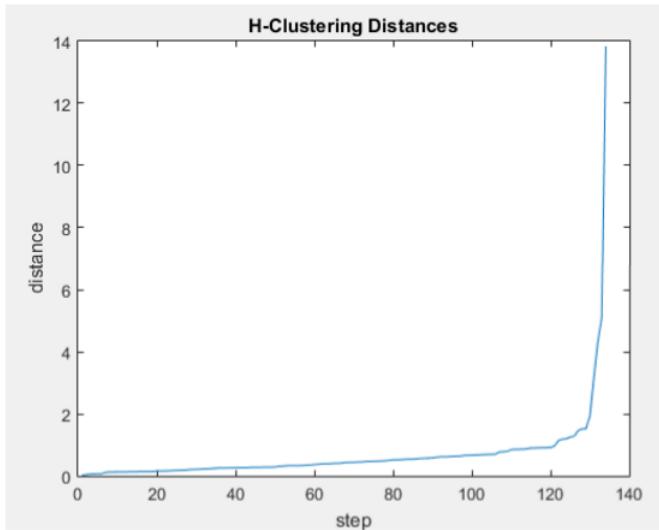


Figure 14.2: Distancias al hacer clustering Jerárquico

Como podemos ver en cada paso la distancia es cada vez mayor, esta información es valiosa. Podemos por ejemplo discretizar las distancias en una cierta cantidad de “buckets” por ejemplo 20 en nuestro ejemplo de 135 puntos. Una vez que hicimos esto podemos encontrar el primer ”hueco” es decir el primer bucket que no contiene ninguna distancia y esto nos da el primer ”gap” o salto de distancias.

En el caso de nuestros 135 puntos el valor es 4 lo cual nos indica que, teóricamente, tenemos 4 clusters en nuestro set de datos. Esto coincide con lo que vemos visualmente. El método no es infalible pero puede usarse como una herramienta ya que no necesita absolutamente ningún parámetro.

Podemos ver ahora que resultado tendríamos si cortamos el algoritmo en una cierta cantidad de clusters fija.

Notemos que si bien hay 4 clusters “visibles” en nuestro set de datos recién llegamos al resultado que queremos con 9 clusters ($k=9$) esto se debe por un lado a la forma en la cual calculamos la distancia entre clusters (en este caso por default) y también a que el método es sensible a los outliers, es decir datos que no pertenecen a ningún cluster.

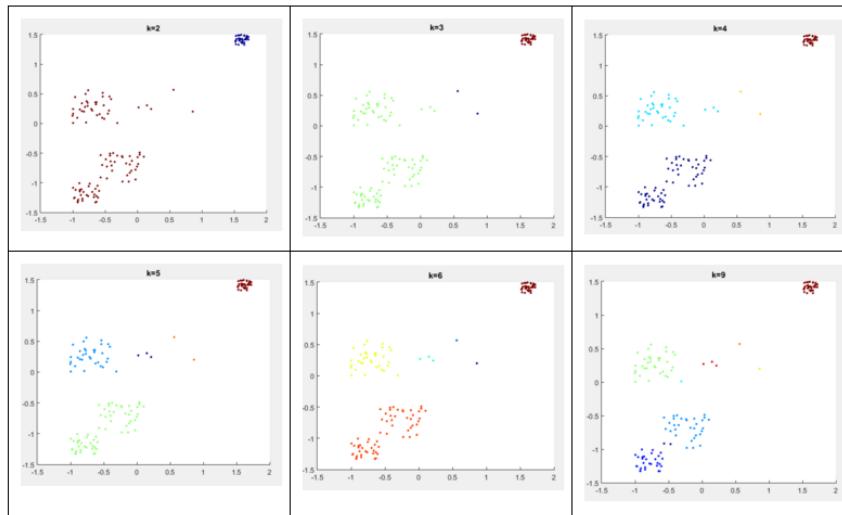


Figure 14.3: Probando la cantidad de clusters

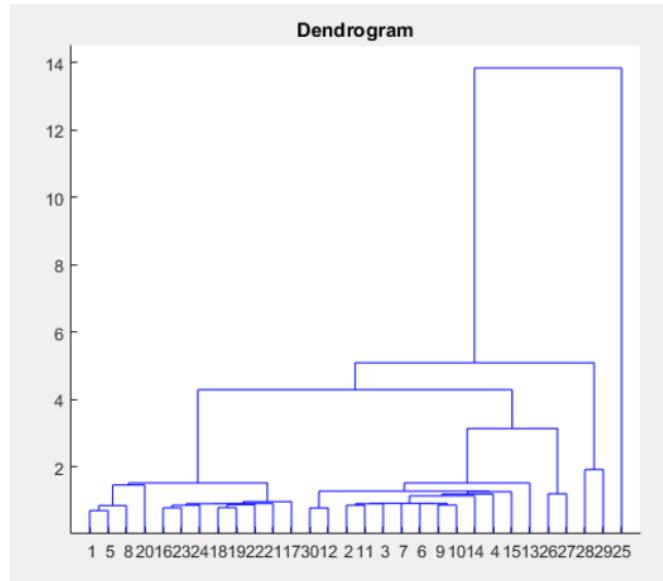


Figure 14.4: Dendrograma

14.1.3 Dendogramas

A partir de los datos generados por la función de clustering jerárquico podemos generar un diagrama que nos muestre en cada paso que clusters se han unido y

con qué distancia. Este diagrama se denomina “dendrograma”.

La altura de las líneas representa la distancia a la cual estaban los puntos. Podemos usar este diagrama para ver que $k = 4$ es mejor que $k = 3$ ya que estaríamos cortando distancias más grandes. Esto nos da una idea aproximada de cuáles pueden ser los valores de k más adecuados para nuestro set de datos.

14.1.4 Performance

Está claro que el método de clustering jerárquico es extremadamente poderoso, tenemos todas las soluciones de clustering posibles para diferentes valores de k al alcance de la mano. Y además tenemos información muy valiosa en la descomposición jerárquica del set de datos.

El problema es cómo programar el algoritmo.

Es necesario en cada paso encontrar los clusters que están a distancia mínima, esto por fuerza bruta implica recorrer los “ n ” clusters que tenemos y por lo tanto nuestro algoritmo es de orden cuadrático. Y esto sin considerar que la distancia entre clusters también puede ser costosa según el método elegido.

La solución pasa por aproximar el problema de los clusters más cercanos en cada etapa, en general se puede usar LSH como un algoritmo para evitar tener que calcular la distancia de todos contra todos. De esta forma por cada cluster simplemente buscamos el más cercano entre la lista de candidatos devuelto por LSH. Esto puede usarse calculando la distancia entre clusters como la distancia entre centroides. Cada vez que creamos un nuevo cluster promediamos sus puntos y este centroide sera el punto que hasheamos con LSH insertándolo en nuestra estructura de datos. Luego por cada punto calculamos el más cercano y nos quedamos el par que nos de distancia mínima. La gran ventaja de usar LSH es que buscar el cluster más cercano a uno dado es $O(1)$ en lugar de $O(n)$.

Una solución por fuerza bruta de clustering jerárquico puede ser mejor que K-Means, una solución aproximada usando LSH en general puede ser mejor, igual o peor que K-Means y por eso es que K-Means suele ser el método de clustering más usado. De todas formas para pocos puntos o como método exploratorio es fundamental conocer la herramienta de clustering jerárquico. Un dendrograma, el plot de distancias o el método para encontrar “ k ” pueden resultar fundamentales para el análisis de varios conjuntos de datos.

14.2 K-Means

K-Means es el mas importante de los algoritmos de clustering, no solo como algoritmo de clustering sino como parte de otros algoritmos. Es un algoritmo que nunca funciona muy mal y a veces funciona muy bien lo cual lo hace una excelente herramienta de propósito general. Si agregamos que no necesita de ningún hiper parámetro excepto la cantidad de clusters podemos comprender porque es tan popular. Vamos a empezar planteando el objetivo de K-Means y luego el algoritmo y sus diferentes variantes y aplicaciones. Es realmente sorprendente la cantidad de usos impensados que podemos tener para un algoritmo

tan sencillo.

14.2.1 El Problema General

El problema de clustering general recibe una serie de m puntos en n dimensiones. El objetivo es encontrar cómo repartir los m puntos en k clusters de la mejor forma posible. En K-Means la cantidad de clusters k es siempre un hiper-parámetro, luego hablaremos de como encontrar el óptimo.

Es necesario, entonces, aclarar que queremos decir con “la mejor forma posible”. Podemos definir a cada cluster a partir de su centro (centroide) en cuyo caso la mejor distribución posible es aquella que minimiza la distancia entre cada punto y el centroide que se le ha asignado. Vamos a trabajar aquí con la distancia euclidiana ya que es la que se utiliza con mayor frecuencia en K-means.

$$\min \sum_{i=1}^m \|x_i - c_{idx(i)}\|^2 \quad (14.1)$$

Esta fórmula calcula la distancia desde cada punto de nuestro set de datos hasta el centroide que le corresponde. $idx(i)$ es una función que nos dice el número de centroide (cluster) asignado al punto i y c_j es la posición del centroide j . Los centroides tienen igual cantidad de dimensiones que los puntos y pueden o no coincidir con puntos de los datos. Cuando un centroide es también un punto de los datos lo llamamos “clusteroide”, en general no vamos a pedir que los centroides sean también puntos sino que vamos a permitir que tomen cualquier posición dentro del espacio de los datos.

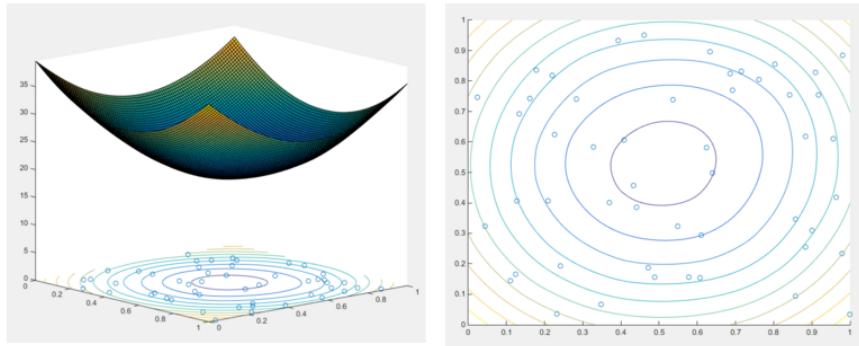
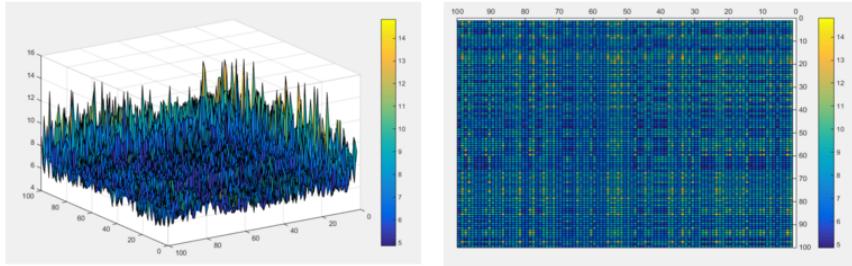
Es importante entender esta función para comprender el problema genérico de clustering.

Sabemos que hay k centroides, cada uno de estos centroides puede estar en cualquier punto de nuestro espacio, el objetivo es encontrar la posición óptima para nuestros centroides de forma tal de minimizar la distancia total entre los puntos y los centroides que se le han asignado. Es evidente que cada punto debe estar asignado a su centroide más cercano para minimizar la distancia por lo que el problema puede resumirse a encontrar la posición óptima para los k centroides.

Si tenemos un conjunto de puntos en dos dimensiones y un solo centroide ($k = 1$) es evidente que la función de distorsión va a tener un único mínimo ya que hay una sola posición para el centroide que minimiza la suma de las distancias al cuadrado entre el centroide y todos los puntos.

Buscar la posición óptima del centroide en este caso es una tarea sencilla. Lamentablemente $k=1$ es el único caso en el cual el problema de clustering genérico es sencillo, apenas pasamos a $k=2$ el problema cobra vida y se convierte en un monstruo.

Usando $k = 2$ y el mismo conjunto de puntos anterior graficamos la función de distorsión en base a la posición de los centroides. Cada posición posible (x, y) la indexamos con un número de $1aN$ de forma tal de poder graficar la función en tres dimensiones:

Figure 14.5: Función costo de K-Means para $k = 1$ Figure 14.6: Función costo de K-Means para $k = 2$

Como podemos ver la función de distorsión tiene ahora muchísimos mínimos locales, esto dificulta muchísimo construir algoritmos que encuentren el mínimo global. De la misma forma podemos observar algo positivo que es que los mínimos locales son muy cercanos al mínimo global por lo que un algoritmo que sea localmente óptimo no está lejos de ser globalmente óptimo. Esta propiedad es la que aprovechó Lloyd para desarrollar el algoritmo K-Means. Antes de ver este algoritmo cerremos el problema de clustering general. El problema de K-Means en general es NP-Hard a partir de $k = 2$ en cualquier espacio euclídeo a partir de \mathcal{R}^2 .

14.2.2 El Algoritmo de Lloyd (conocido como K-Means)

El algoritmo de Lloyd, que es el que conocemos como K-Means, es una forma de aproximar el problema de clustering general. Una de sus grandes virtudes es que se trata de un algoritmo muy sencillo que consta de solamente un ciclo y dos operaciones.

Lo que hacemos es simplemente inicializar k centroides al azar, por ejemplo

Algorithm 27: Kmeans

```

1 Inicializar Centroides
2 while not convergence do
3   | asignarle a cada punto su centroide mas cercano
4   | recalcular cada centroide como el promedio de todos sus puntos

```

eliendo k puntos del set de datos. Luego asignamos a cada punto a su centroide mas cercano. Re-calculamos los centroides como el promedio de todos sus puntos y repetimos el proceso hasta que los centroides convergen.

K-Means pertenece a la familia de algoritmos EM (Expectation Maximization), partiendo de una cierta posición para los centroides la fase “E” (expectation) asigna a cada punto el centroide mas cercano. La fase “M” (maximization) recalcula cada centroide de forma tal de minimizar la distancia desde el centroide a los puntos asignados al mismo.

La convergencia puede verificarse mediante la diferencia entre los centroides entre el paso anterior y el actual, cuando los centroides prácticamente ya no cambian de posición declaramos la convergencia del algoritmo.

Podemos garantizar que K-Means converge si analizamos la función de distorsión o costo:

$$J(c, idx) = \sum_{i=1}^m \|x_i - c_{idx(i)}\|^2 \quad (14.2)$$

En la fase “E” los centroides están fijos y al asignarle a cada punto su centroide más cercano J disminuye. En la fase “M” se asigna a cada centroide el promedio de sus puntos y eso también hace que J disminuya. Por lo tanto J es monótona y decreciente lo cual garantiza su convergencia a un mínimo local.

Veamos un ejemplo:

Como podemos ver hay unos 4 clusters en nuestros datos y varios puntos aislados. Vamos a probar entonces correr K-Means con $K=4$. Los centroides los vamos a inicializar al azar dentro del rango $[-3..4, -2..5]$.

A continuación mostramos dos ejecuciones de K-Means inicializando los centroides al azar.

Las líneas muestran el movimiento de los centroides desde su posición inicial (random) hasta la posición en la cual el algoritmo converge. Podemos ver que en la primera ejecución (izquierda) K-Means detecta bien el cluster amarillo y el verde. Pero el cluster rojo es en realidad dos clusters y el azul reúne a un conjunto de puntos que no están asociados. En la segunda ejecución (derecha) el algoritmo detecta correctamente los cuatro clusters que queríamos identificar.

La conclusión es muy simple: al ser un algoritmo que encuentra un mínimo local K-means es muy sensible a la posición inicial de los centroides.

Una posible solución para no caer en un mínimo local malo es realizar varias ejecuciones de K-means, idealmente en paralelo, y computar la función de distorsión “ J ” para cada resultado final quedándonos con la mínima. Notemos que

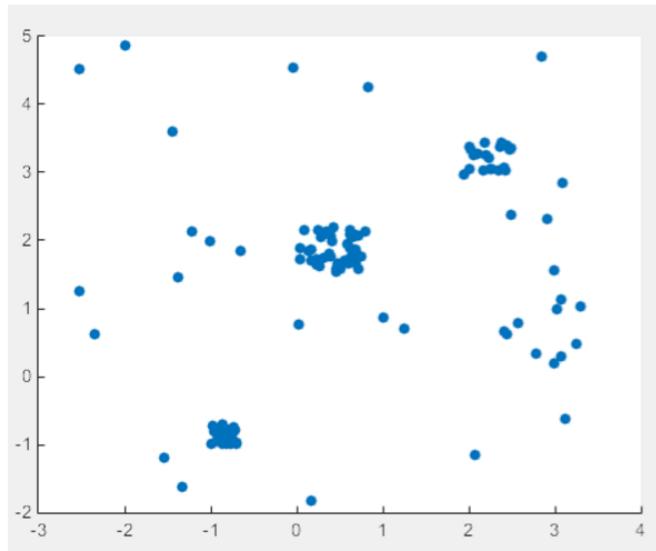


Figure 14.7: Ejemplo K-Means

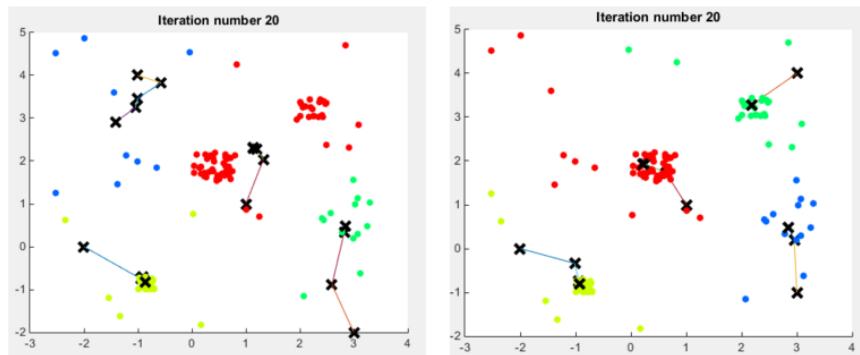


Figure 14.8: Ejemplo K-Means

esto es una aplicación directa de un algoritmo tipo Montecarlo en donde en cada iteración probamos una inicialización aleatoria para los centroides.

Veamos la mejor y peor solución de K-Means:

Como podemos ver el resultado de mejor costo es lo que esperábamos mientras que el resultado de peor costo es particularmente terrible.

El siguiente gráfico muestra el resultado de la función costo para 100 ejecuciones de K-Means y un histograma representando la cantidad de veces que observamos diferentes rangos de costo.

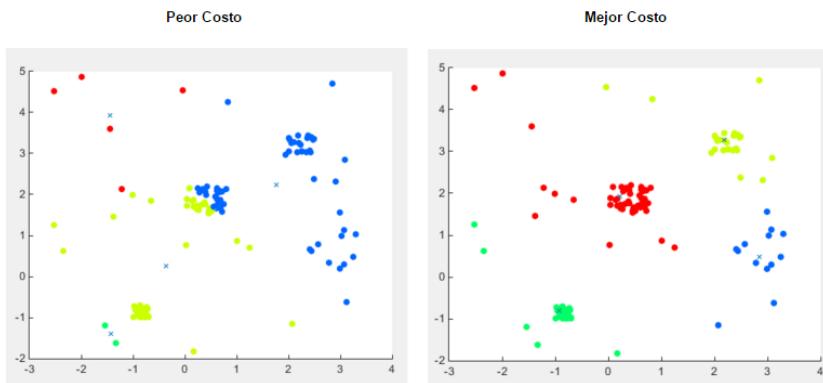


Figure 14.9: Ejemplo K-Means mejor y peor costo

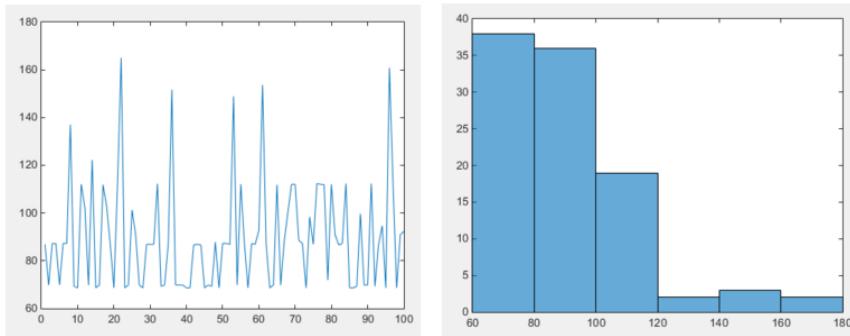


Figure 14.10: Costo en K-Means

Como podemos ver, afortunadamente los costos bajos son amplia mayoría, es decir que únicamente con algunas pocas inicializaciones desafortunadas K-means llega a un mínimo global muy malo. Esto quiere decir que una sola iteración de K-Means nos dará un buen resultado con una probabilidad alta y hacer un ciclo con unas pocas ejecuciones es más que suficiente para llegar a un excelente resultado.

Veamos ahora si inicializando los centroides de otra forma podemos garantizar un buen resultado en una única ejecución.

14.2.3 K-Means++

K-Means++ [Arthur '07] es una variante de K-Means en donde lo único que cambia con respecto al algoritmo de Lloyd es la forma en que se inicializan los centroides. La idea de K-means++ es asignar los centroides de forma espaciada,

de esta forma el óptimo local obtenido por K-Means tiene una mayor probabilidad de estar cerca del óptimo global; o lo que es lo mismo el óptimo local está lejos de los óptimos que son muy malos con una probabilidad alta.

Algorithm 28: Kmeans++

- 1 Elegir un punto al azar como primer centroide
 - 2 **for** $i=2:k$ **do**
 - 3 por cada punto calcular la distancia con cada centroide y quedarnos el mínimo
 - 4 calcular la probabilidad de cada punto como su distancia mínima dividido la suma de todas las distancias mínimas
 - 5 elegir un nuevo punto al azar en base a estas probabilidades
 - 6 agregar el punto a la lista de centroides
-

Supongamos que el primer centroide elegido al azar es k_1 . El objetivo ahora es encontrar el siguiente centroide k_2 de forma tal que esté alejado de k_1 . Pero no elegimos simplemente el centroide más lejano. Lo que hacemos es asignar a cada punto una probabilidad de acuerdo a su distancia a k_1 y en base a dicha probabilidad elegimos el próximo centroide al azar. Notar que si ya tuviéramos más de un centroide entonces la probabilidad la calcularíamos en base a la distancia de cada punto al centroide más cercano.

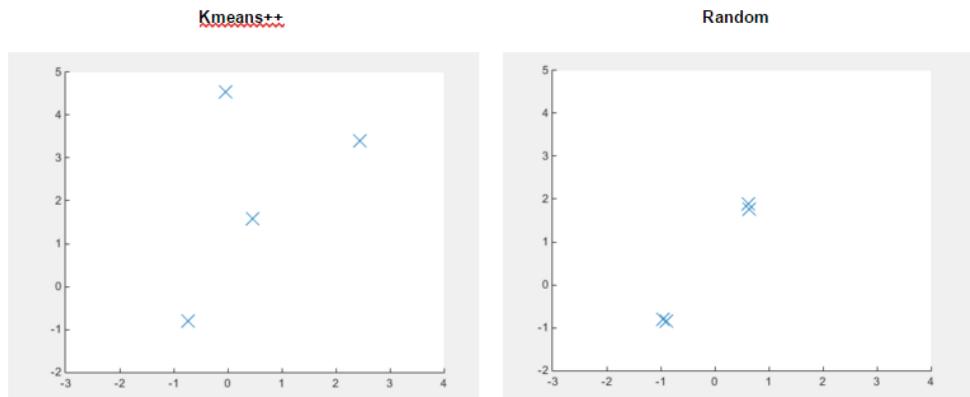


Figure 14.11: Kmeans++

Como podemos ver K-means++ genera una inicialización más dispersa de los centroides iniciales. El problema de K-Means++ es que para datos realmente masivos necesita hacer " k " iteraciones sobre los datos para elegir los centroides, tarea que es todavía más ineficiente cuando " k " es un número largo. Existe una variante a K-Means++ [Bahmani] llamada K-Means paralell. La idea es hacer

solo $\log(k)$ pasadas sobre los datos, en cada iteración se agregan l centroides en lugar de 1, el resultado es que tendremos una cantidad de centroides mayor a k que son los centroides que necesitamos, los centroides finales se obtienen clusterizando los candidatos a centroides ponderados por la probabilidad con la cual fueron seleccionados, este paso se puede hacer, por ejemplo, usando clustering jerárquico.

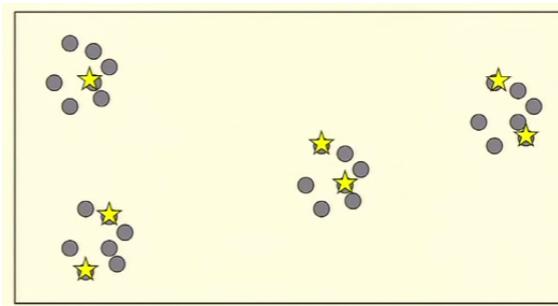


Figure 14.12: Suponiendo que usamos $k = 4$ y $l = 3$ en el primer paso elegimos un punto al azar como centroide, en el segundo paso elegimos 3 puntos usando K-means++, en el tercer paso otros 3 puntos y tenemos en total 7 candidatos a centroides. Clusterizando estos 7 centroides hasta que nos queden 4 nos quedarían los 4 centroides a usar como inicialización de K-Means



Figure 14.13: Resultado luego de clusterizar los 7 centroides en 4 clusters.

En general K-Means++ y su variante K-Means parallel generan una inicialización mejor para K-Means y esto permite no solo obtener una mejor solución final sino que también acelera la velocidad de convergencia del algoritmo.

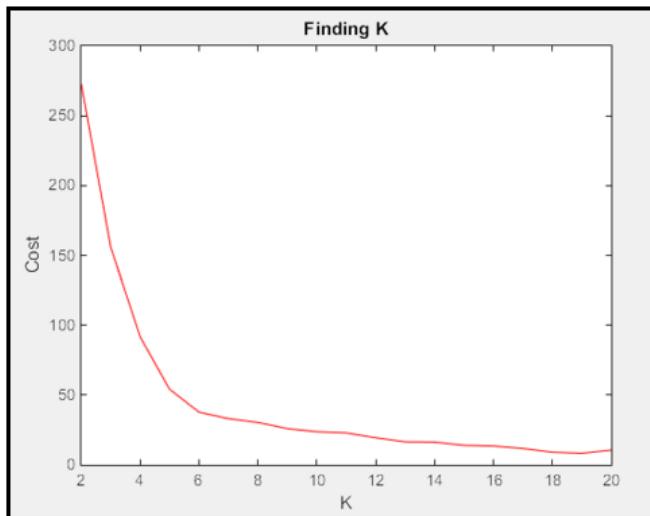


Figure 14.14: Encontrando K

14.2.4 Como buscar K

En muchas ocasiones el número de clusters k a generar es desconocido. Es necesario entonces encontrar el k óptimo. En general la forma de buscar k es mediante un Grid-Search con diferentes valores posibles.

Como métrica para evaluar la calidad de cada valor de k podemos usar un promedio de nuestra función de distorsión para varias iteraciones de K-Means o el diámetro promedio de los clusters, tomando como diámetro de un cluster la distancia máxima entre dos puntos del mismo. Existen otras métricas para evaluar la calidad de una solución de clustering pero todas tienen alguna falla así que es conveniente optar por la más simple.

Podemos graficar la función costo en función de K .

El k óptimo en general está en el “codo” de este gráfico. En nuestro caso corresponde a $k=4$ lo cual es bastante lógico ya que teníamos cuatro clusters naturales. Es de destacar que cuando k es igual a la cantidad de puntos el costo total es cero y cada cluster contiene solo un punto pero esto obviamente no tiene mucho sentido.

14.2.5 Diagramas de Voronoi

El proceso de clustering de K-means genera lo que llamamos un diagrama de “Voronoi” o una teselación de Voronoi que consiste en delimitar la frontera hasta la cual llega cada centroide. Por ejemplo para nuestro set de datos ejemplo corriendo K-Means nos queda delimitado el siguiente diagrama de Voronoi:

El diagrama de Voronoi nos muestra a qué cluster quedaría asignado un

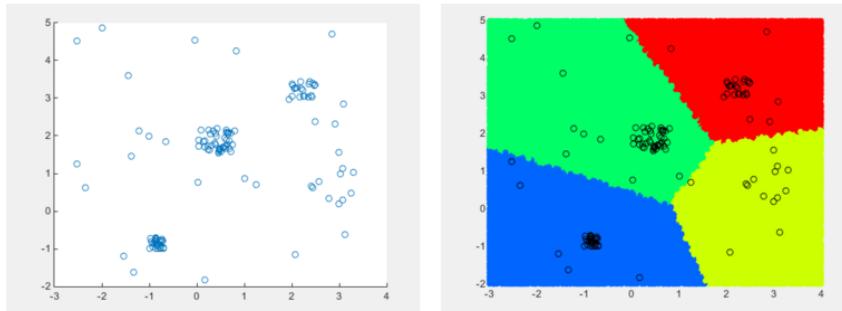


Figure 14.15: Diagrama de Voronoi

punto nuevo que quisiéramos agregar a nuestros datos sin correr de nuevo K-Means. Un detalle fundamental de esto es que, como podemos ver, K-means realiza una partición de todo el espacio n-dimensional en el cual trabajamos. Esto quiere decir que podemos representar a cualquier punto de nuestro espacio de n-dimensiones en k-dimensiones de acuerdo a la separación establecida por el diagrama de Voronoi correspondiente al resultado de K-means. Esto va a ser muy útil más adelante.

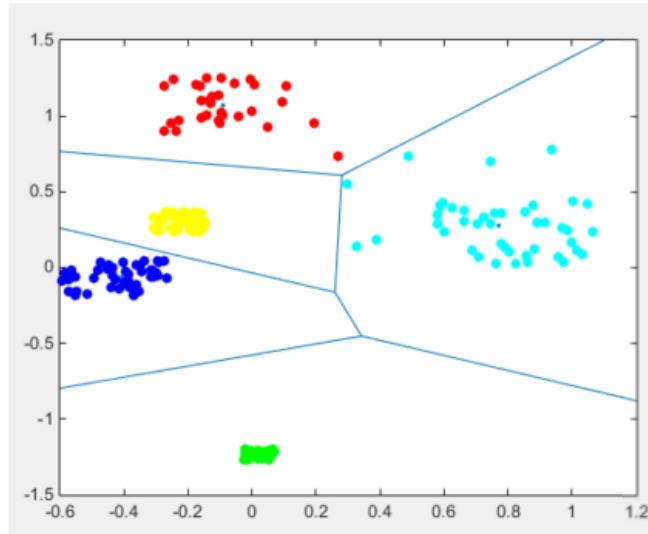


Figure 14.16: Diagrama de Voronoi

Notemos la forma que toma un diagrama de Voronoi, cada sector del diagrama depende única y exclusivamente de un punto: el centroide. Las fronteras

indican la separación entre centroides, a cada lado de las líneas estamos mas cerca de uno u otro centroide. El tamaño de cada área no depende de la cantidad de puntos asociados al centroide sino de que tan aislado se encuentra el centroide respecto del resto, cuanto mas aislado mas grande será la superficie de su área.

14.2.6 Kmeans Online

Cuando se trata de procesar datos masivos, millones o billones de datos, el principal problema de K-Means es que requiere en cada iteración calcular el centroide más cercano a cada uno de los puntos y esta operación es costosa ya que hay que comparar cada punto contra cada centroide lo cual implica visitar todos los puntos del set de datos por cada iteración.

La versión online de K-Means procesa un punto por vez del set de datos, no necesita conocer todos los puntos antes de empezar y requiere un mínimo uso de memoria.

La idea es muy sencilla a partir de “k” centroides inicializados de alguna forma que es externa al algoritmo, por ejemplo al azar, se procesa cada punto asignándolo al centroide más cercano y luego se mueve ese centroide hacia el punto procesado de forma proporcional a la cantidad de puntos que tiene el cluster en total. Es necesario contar con un vector de “k” elementos en donde vamos a ir contando cuantos puntos contiene cada cluster hasta el momento.

Algorithm 29: K-Means Online

```

1 n[1..k] = 0
2 Asignar k centroides al azar  $c_1..c_k$ 
3 for  $i=1:m$  do
4   sea  $j$  el centroide mas cercano a  $x_i$ 
5   idx(i)=j
6   n[j]++
7    $c_j = c_j + \frac{1}{n[j]}(x_i - c_j)$ 

```

Es importante destacar que por cada punto el algoritmo computa la distancia contra cada centroide pero la cantidad de centroides nunca es demasiado grande, el algoritmo tradicional, en cambio, computa en cada iteración la distancia de todos los puntos a todos los centroides.

Pueden realizarse varias iteraciones de OnlineKMeans en caso de ser necesario, aunque cuando los puntos son millones o provienen de un stream infinito suele ser necesario y suficiente realizar una única iteración.

Es interesante analizar el comportamiento del algoritmo frente a un stream, un flujo de datos infinito, a medida que se procesan mas y mas puntos cada punto va a quedar asociado mediante su cluster a aquellos que son más cercanos. Esto permite realizar clustering y clasificación online sobre streams sin prácticamente ninguna restricción de tiempo ni memoria! Veamos el diagrama de Voronoi

resultante luego de procesar miles de puntos en 2D

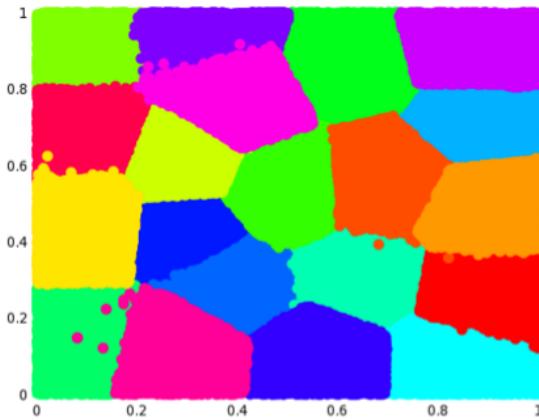


Figure 14.17: Kmeans Online

Podemos ver que el diagrama es muy similar al diagrama de Voronoi de K-Means tradicional con algunos puntos aislados que quedan dentro de un cluster pero pertenecen a otro. Esto es fácil de explicar ya que estos puntos fueron asignados originalmente a un cluster cuyo centroide se fue moviendo luego, la versión online de K-Means únicamente asigna una vez por punto, una vez que el punto fue asignado a un centroide permanece dentro de dicho cluster. Esto puede solucionarse, por supuesto, realizando mas de una iteración.

En la segunda iteración los puntos que quedaron mal asignados en la primera pasada pasan al cluster que les corresponde.

En el siguiente experimento promediamos varias ejecuciones de K-Means y K-Means Online para diferente número de iteraciones y comparamos el costo.

Como es lógico el costo promedio de la versión online disminuye a medida que aumentamos las iteraciones alcanzando su mejor rendimiento en unas 10 iteraciones mientras que la versión tradicional oscila según comience con mejores o peores inicializaciones. Lo mas destacable es que a partir de tres o cuatro iteraciones el resultado de la versión online es siempre mejor que la versión tradicional.

La versión Kmeans++ es mejor que las dos anteriores aunque muy similar a la versión online con múltiples iteraciones con el costo de tener que pre-procesar los datos para lograr los centroides iniciales.

Teoría de K-Means Online

Vamos a probar ahora que K-Means online también optimiza la función costo de K-Means [Bottou-Bengio].

Podemos considerar la función costo de K-Means como la sumatoria del costo de cada punto.

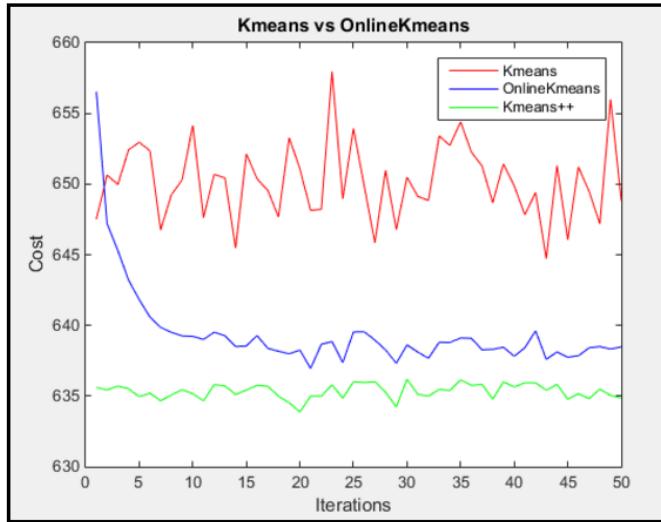


Figure 14.18: Kmeans Online vs Kmeans++ y Kmeans

$$J = \sum_i L(x_i, c) \quad (14.3)$$

En donde el costo de cada punto es su distancia al centroide mas cercano:

$$L(x_i, c) = \min_k \frac{1}{2} (x_i - c_k)^2 \quad (14.4)$$

El Hessiano de la función costo es la matriz que tiene la derivada segunda de la función costo con respecto a cada par de parámetros (centroideos). Podemos descomponer la matriz H como la suma de matrices H_i en donde H_i es la matriz asociada al i-ésimo punto. Además H_i se puede descomponer en bloques para cada par de centroides el costo $L(x_i, c)$ solo depende del centroide mas cercano al punto x_i por lo tanto ese bloque de H_i es la matriz identidad y el resto son todos ceros. Si sumamos todas las matrices H_i lo que obtenemos es una matriz diagonal en donde cada elemento de la diagonal es la cantidad de puntos asociados a cada centroide:

$$H = \begin{pmatrix} N_1 & 0 & \dots & 0 \\ 0 & N_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & N_k \end{pmatrix}$$

Por lo tanto aplicando el método de Newton-Raphson la regla de actualización de los centroides es:

$$\Delta c_k = \sum_i \begin{cases} \frac{1}{N_k} (x_i - w_k) & \text{if } x_i \in k \\ 0 & \text{otherwise} \end{cases}$$

Notemos que esta es precisamente la definición de K-Means online por lo tanto podemos afirmar que K-Means online optimiza la función costo de K-Means mediante el método de Newton-Raphson!

14.2.7 Soft K-Means

En esta sección vamos a ver una generalización de K-Means que se conoce con varios nombres: Soft K-Means, Fuzzy C-Means o simplemente EM (Expectation Maximization). El objetivo de este algoritmo es muy simple: permitirle a cada punto pertenecer al mismo tiempo a más de un cluster. Generalizando, cada punto pertenece a todos los clusters y lo que determinamos en el algoritmo es el grado de pertenencia de cada punto a cada cluster o la probabilidad de que el punto pertenezca a cada cluster.

El algoritmo K-Means tal y como lo conocemos asigna a cada punto al centroide más cercano realizando luego el recálculo de los mismos, eventualmente el algoritmo converge a la asignación final en la cual los centroides ya no se mueven. Decimos que la asignación de K-Means es “hard” ya que cada punto queda asignado a un único cluster, el más cercano y la posición de todos los demás centroides no es importante.

Una versión “soft” de K-Means asigna a cada punto k probabilidades, siendo cada una de ellas la probabilidad de que el punto se asigne a cada uno de los k -centroides. Cuanto más cerca está el punto del centroide mayor es su probabilidad. Para calcular estas probabilidades podemos sumar todas las distancias y luego realizar $1-(\text{dist}/\text{suma})$. De esta forma la suma de todas las probabilidades para cada punto es igual a 1.

El algoritmo se basa en dos pasos que son los mismos que teníamos en K-Means.

1. Dada la asignación de cada punto a cada cluster calcular los centroides.
2. Dados los centroides calcular la asignación de cada punto a cada cluster.

Estos procesos se repiten en un ciclo hasta que los centroides no cambien (con una cierta tolerancia).

Para calcular la asignación de cada punto “ i ” a cada cluster “ k ” usamos la siguiente fórmula:

$$w_{ik} = \left(\sum_j \left(\frac{d(c_k, x_i)^2}{d(c_j, x_i)^2} \right) \right)^{-1} \quad (14.5)$$

Notemos que la fórmula simplemente calcula la distancia entre el punto y el centroide dividiendo por la sumatoria de la distancia entre el punto y todos los centroides. Finalmente elevando esto al cuadrado.

Para calcular los centroides usamos la siguiente fórmula:

$$c_k = \frac{\sum_i x w_{xk}^2}{\sum_x w_{xk}^2} \quad (14.6)$$

Es decir simplemente un promedio de todos los puntos ponderado por el peso de la asignación de cada punto a cada centroide.

Veamos un ejemplo:

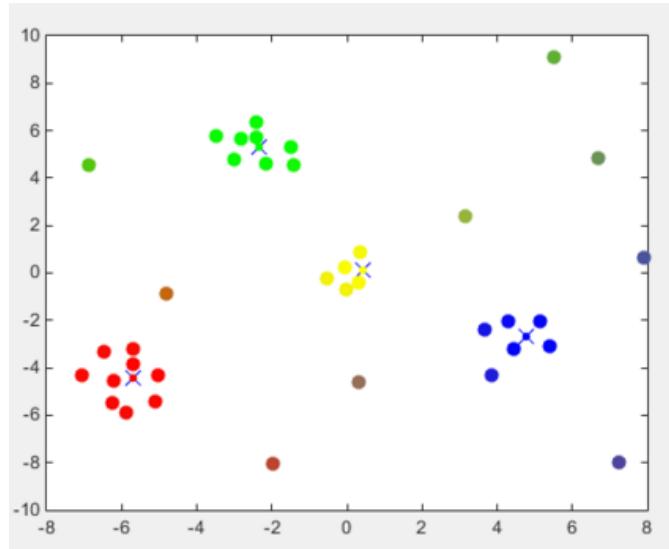


Figure 14.19: Soft Kmeans

El resultado nos muestra una típica salida de K-Means los puntos cercanos a los centroides están asignados casi por completo a los mismos y los puntos “en el medio” están repartidos. Hay un punto de color violeta en el medio de los clusters rojo y azul. El punto de más arriba de todo es mayormente verde pero tiene algo de azul. Etc.

Podemos generar el diagrama de Voronoi para la versión ”soft” de K-Means:

Y este diagrama de Voronoi “soft” nos muestra claramente el grado de influencia de cada cluster en nuestro espacio de puntos. Esto es algo muy interesante además de verse muy bonito.

Versión Online

Al igual que con la versión tradicional de K-Means podemos necesitar una versión online para poder procesar millones o billones de puntos. Vamos a proceder exactamente igual que en la versión tradicional. Tomamos un punto, calculamos la probabilidad de cada centroide en base al punto y acercamos a cada centroide en proporción a dicha probabilidad y a la acumulación total de probabilidades que tenga el centroide. Es decir que cuantos más puntos se

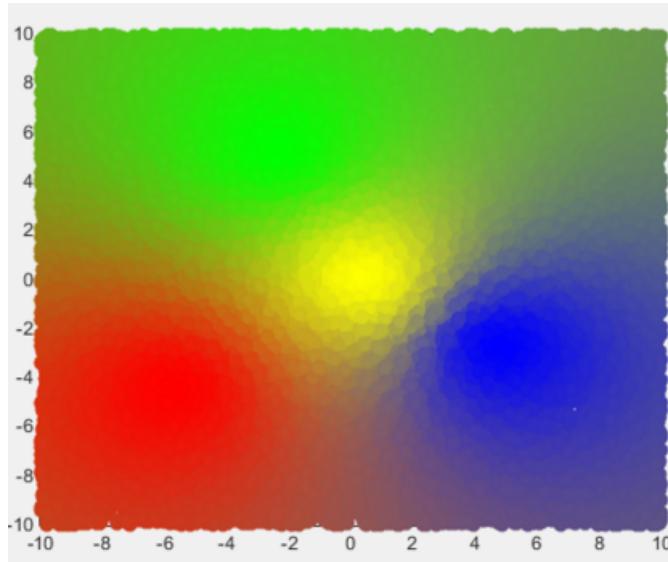


Figure 14.20: Soft Kmeans

“peguen” fuertemente a un centroide menos se moverá el mismo al procesar un punto nuevo.

La versión online de Soft K-Means tiende a generar el mismo resultado que la versión anterior en una cantidad relativamente baja de iteraciones.

La versión “soft” de K-Means es una generalización en la cual obtenemos para cada punto la probabilidad de que el mismo pertenezca a cada cluster. Esto tiene una enorme cantidad de aplicaciones ya que en muchísimos problemas de clustering una asignación “soft” de los puntos es preferible. Por ejemplo si estamos haciendo clustering de documentos de texto es posible que un texto pertenezca a varios “temas” en distinto nivel. Por ejemplo 20% noticias, 80% deportes. Así como éste podemos pensar en muchos ejemplos más. Soft K-Means es un algoritmo muy importante y que debemos tener siempre presente.

14.2.8 La base de Kmeans y sus aplicaciones

Para entender esta sección vamos a partir de un ejemplo de Soft K-Means. Vamos a usar 4 puntos en 3 dimensiones que representamos con la siguiente matriz de 4x3:

$$X = \begin{pmatrix} 0.815 & 0.632 & 0.957 \\ 0.906 & 0.097 & 0.965 \\ 0.127 & 0.278 & 0.158 \\ 0.913 & 0.547 & 0.970 \end{pmatrix}$$

Vamos a aplicar Soft-Kmeans usando $k = 2$, como resultado vamos a tener

por un lado el grado de pertenencia de cada punto a cada cluster que podemos representar con una matriz de 4x2:

$$IDX = \begin{pmatrix} 0.0328 & 0.9672 \\ 0.0847 & 0.9153 \\ 0.9999 & 0.0001 \\ 0.0096 & 0.9904 \end{pmatrix}$$

Según vemos el punto 3 pertenece a un cluster y el resto pertenecen a un segundo cluster. Mirando los puntos esto tiene sentido ya que el tercer punto tiene la primer y tercer coordenada completamente a los otros 3 que tienen valores muy similares!

Otro resultado que nos da Soft Kmeans son los centroides, como $k = 2$ vamos a tener 2 centroides de 3 dimensiones:

$$C = \begin{pmatrix} 0.1333 & 0.2776 & 0.1643 \\ 0.8776 & 0.4392 & 0.9644 \end{pmatrix}$$

El primer centroide es casi igual al punto 3 y el otro centroide es el promedio ponderado de los otros 3 puntos de acuerdo a sus probabilidades de pertenecer a cada cluster.

Lo interesante de todo esto es que podemos tomar a los centroides como una *base* y a las asignaciones como *coordenadas* y entonces nuestra matriz de puntos X se puede aproximar como $IDX * C$

$$\begin{pmatrix} 0.0328 & 0.9672 \\ 0.0847 & 0.9153 \\ 0.9999 & 0.0001 \\ 0.0096 & 0.9904 \end{pmatrix} \begin{pmatrix} 0.1333 & 0.2776 & 0.1643 \\ 0.8776 & 0.4392 & 0.9644 \end{pmatrix} = \begin{pmatrix} 0.8532 & 0.4340 & 0.9382 \\ 0.8145 & 0.4256 & 0.8966 \\ 0.1334 & 0.2776 & 0.1643 \\ 0.8704 & 0.4377 & 0.9567 \end{pmatrix}$$

Notemos que la matriz reconstruida es muy similar a la matriz original, la tercera fila que corresponde al tercer punto de X es prácticamente igual. En el caso de K-Means tradicional el funcionamiento es igual pero la matriz IDX tiene únicamente 1s y 0s porque la asignación es hard pero igual podemos usar esta matriz y los centroides para aproximar la matriz original, en el caso de Kmeans tradicional cada punto se representa mediante su centroide. En el caso de Soft K-Means cada punto se representa mediante una combinación lineal de todos los centroides. Todo esto nos lleva a conclusiones que son realmente muy importantes:

1. Podemos usar K-Means como una forma de reducción de dimensiones, en donde cada punto se representa mediante las coordenadas del mismo con respecto a una base que son los centroides.
2. La cantidad de clusters k nos da la cantidad de vectores en la base y por lo tanto nos permite regular la cantidad de dimensiones en la que representaremos a los datos.

3. El diagrama de Voronoi nos da las fronteras para representar nuevos puntos de acuerdo a la base obtenida originalmente por K-Means.
4. Cada vector de la base es un centroide por lo tanto es un promedio de puntos de nuestro set de datos
5. Por lo tanto usando K-Means como base representamos a cada punto mediante una combinación lineal de vectores que surgen de promediar puntos de nuestro set de datos.
6. Por ser promedio de puntos del set de datos los vectores de la base de Kmeans pueden interpretarse como nuevos datos.
7. A mayor cantidad de centroides mas precisa será la reconstrucción de la matriz de datos original.
8. Cada punto se representa mediante una combinación afín de centroides (las coordenadas son positivas y suman 1) es decir que por cada centroide tomamos un cierto porcentaje del mismo y lo usamos para representar parte de nuestro punto. Notar que esto vale tanto para soft K-Means como para hard K-Means en cuyo caso hay un único centroide con coordenada 1 y todos los demás son 0.

A modo de ejemplo veamos qué pasa si aplicamos K-Means con $k = 10$ al set de datos Mnist que tiene 60000 imágenes en 784 dimensiones.



Figure 14.21: Centroides de Kmeans con $k = 10$ para MNIST

Como podemos ver K-Means encuentra centroides que representan cada "cluster". Lo ideal sería que cada cluster represente a un dígito pero este caso sería demasiado perfecto. Podemos ver que hay una cierta confusión entre los "4"s y los "9"s ya que hay tres centroides que parecen "9"s. Si repetimos el proceso usando ahora 100 centroides obtenemos lo siguiente:

Estos son los 100 centroides encontrados por K-Means como vemos tenemos varias formas de escribir cada dígito representada por un centroide. Estos centroides son promedios de los puntos asignados a los mismos. Podemos ver que hay diferentes formas de escribir cada dígito. Para MNIST tal vez 100 centroides sean muchos pero siempre es interesante ver los centroides que genera K-Means e interpretarlos como si fueran datos.

Figure 14.22: Centroides de Kmeans con $k = 100$ para MNIST

14.2.9 Reducción de Dimensiones con K-Means

Habiendo hablado de la base de K-Means podemos ver como usar K-Means para reducir las dimensiones de un set de datos. Una posibilidad es representar a cada punto mediante su centroide mas cercano (hard Kmeans) o mediante una combinación afín de centroides. Los sistemas basados en coordenadas requieren de la base de K-Means para tener sentido, el objetivo es encontrar una forma de representar los datos en k dimensiones sin que sean necesarios los centroides.

Una forma simple de hacer esto es por cada punto calcular la distancia a cada uno de los k centroides y usar estas k distancias como k coordenadas por cada punto. Esto nos da una forma simple y directa de representar los datos en k dimensiones.

Para corregir problemas de escala podemos usar una función que mapee la distancia al centroide x a un número entre 0 y 1 por ejemplo:

$$y = e^{-x\gamma} \quad (14.7)$$

γ es un hiper-parámetro que regula la forma en la que mapeamos, por ejem-

plo con $\gamma = 0.5$ tenemos:

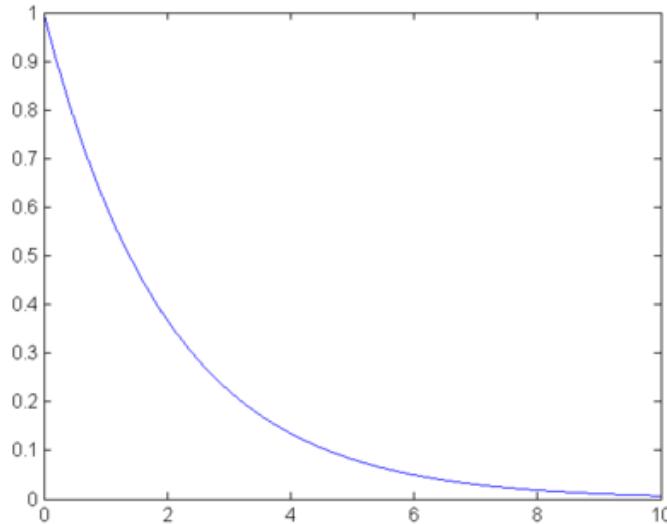


Figure 14.23: Gamma map con $\gamma = 0.5$

En este caso las distancias mayores a 8 van a quedar todas muy cerca de 0. El objetivo es encontrar un γ que sea eficiente para distribuir las distancias que tenemos entre los puntos y los centroides de forma pareja. Esto puede hacerse, por supuesto usando grid-search y luego un histograma de los valores que tenemos para las k coordenadas de nuestros puntos buscando que la distribución tienda a ser normal. Tener "features" con distribución normal es algo terriblemente beneficioso para varios algoritmos y podemos lograrlo usando K-Means.

Otra opción es simplemente usar un corte no-lineal para las distancias de los puntos a los centroides:

$$y = \max(0, x - \text{avgdist}) \quad (14.8)$$

Es decir que calculamos la diferencia entre la distancia y el promedio de distancias y si nos da negativo la convertimos en 0.

Ambas opciones permiten extraer " k " features a partir de un set de datos en cualquier cantidad de dimensiones luego de aplicar K-Means con " k " centroides. Es muy sorprendente que estos features aprendidos por K-Means funcionan realmente muy bien como representación del set de datos, es muy común que un algoritmo de clasificación funcione mejor con estos features que con los datos originales.

Dado que K-Means es en general eficiente un método sorprendente para manejar clasificación en un set de datos es aplicar K-Means par reducir los datos a k dimensiones y luego aplicar sobre estos datos un clasificador lineal, hay varias cosas que funcionan muy bien basadas en esta técnica.

14.2.10 Kmeans Esférico

K-Means esférico es la aplicación de K-Means a la distancia angular en lugar de la distancia euclíadiana.

Esta distancia es útil siempre y cuando estemos agrupando datos en base a la dirección hacia la cual los datos apuntan en lugar de su posición en el espacio. Esto es particularmente útil para textos en donde la semejanza entre un texto y otro está directamente relacionada por la distancia angular entre los mismos. Podemos pensar que el “ángulo” de cada texto representa su significado, su semántica por lo que al usar la distancia angular en lugar de la euclíadiana estamos comparando el significado de los textos y no los textos en sí mismos (!).

Este es un concepto muy poderoso ya que más de una vez textos que son completamente diferentes en contenido en realidad tienen el mismo significado.

Recordemos que el coseno del ángulo entre dos vectores se puede calcular como:

$$\cos \theta = \frac{\langle x \cdot y \rangle}{\|x\| \|y\|} \quad (14.9)$$

Cuanto mayor es el coseno mas chico es el ángulo y más parecidos son los vectores. Cuanto menor sea el coseno mayor el ángulo y mayor la distancia entre los vectores.

Si normalizamos los vectores para que tengan norma=1 el coseno del ángulo entre los mismos se reduce al producto interno entre los vectores. Por lo tanto el primer paso en K-Means esférico es normalizar todos los vectores para que tengan norma=1. Esta normalización es por filas, no por columnas como hacemos habitualmente, lo que buscamos es que cada fila de nuestro set de datos tenga norma 1.

Los centroides iniciales los vamos a generar al azar con componentes entre 0 y 1 y también los vamos a normalizar por filas.

Ahora tenemos un conjunto de “m” puntos en “n” dimensiones y tenemos “k” centroides en “n” dimensiones. Para calcular cuál es el centroide más cercano a cada punto tenemos que hacer el producto interno entre cada punto y cada centroide. Podemos hacer todas las cuentas juntas calculando el producto entre la matriz X de mxn y la matriz de centroides de kxn transpuesta.

$$D = X * C \quad (14.10)$$

Esto nos da una matriz de m filas por k columnas. Donde cada fila representa a un punto y cada columna representa la semejanza (coseno) a cada uno de los “k” centroides.

Para determinar a que centroide se asigna cada punto simplemente observamos cual es el valor máximo para los cosenos entre cada punto y cada centroide. Notar que al contrario que en K-Means tradicional aquí estamos tomando el máximo producto interno que corresponde a la mínima distancia angular.

La matriz de asignación de puntos a centroides la vamos a construir de mxk en donde cada fila solo tiene un valor distinto de cero que es el valor del coseno entre dicho punto y su centroide mas similar.

$$S = \max_i D \quad (14.11)$$

Hemos cometido un abuso de notación, lo que queremos decir es que S es la matriz en la cual por cada fila de D dejamos únicamente el valor máximo y el resto los convertimos en ceros.

Ahora podemos recalcular los centroides como un promedio de los puntos que pertenecen a cada centroide ponderado por el valor del coseno del centroide con cada punto. Es decir simplemente calcular:

$$C = S^{(t)} X \quad (14.12)$$

Una variante es hacer:

$$C = C + S^{(t)} X \quad (14.13)$$

Esto se conoce como *dampening* y sirve para que los centroides no cambien tan bruscamente de una iteración a otra. El paso final de K-Means esférico es normalizar los centroides para que tengan norma 1 y podamos volver a usarlos en la próxima iteración.

Algorithm 30: Kmeans Esférico

```

1 X = RowNormalize(X)
2 Asignar C=k centroides al azar
3 C = RowNormalize(C)
4 while not convergencia do
5   | D = X * C(t)
6   | S = (D == max(C))
7   | C = (S(t) * X) + C
8   | C = RowNormalize(C)

```

En general este algoritmo converge rápidamente en 10 o menos iteraciones y es bastante mas veloz que la variante euclídea.

Versión Online

La versión que hemos visto es eficiente pero puede consumir mucha memoria al calcular el producto entre la matriz de puntos y la matriz de centroides. Si los puntos son millones o si tenemos millones de dimensiones (o ambas) el cálculo de la matriz podría ser inviable. Por lo tanto necesitamos una versión “online” del algoritmo que pueda funcionar punto a punto.

Afortunadamente no hace falta hacer casi ninguna modificación excepto agregar un loop que procese punto por punto calculando el producto entre dicho punto y la matriz de centroides. Una vez procesados todos los puntos re-calculamos los centroides e iteramos este procedimiento tantas veces como sea necesario.

Voronoi Esférico

Podemos ver que tipo de forma tiene el diagrama de Voronoi asociado cuando usamos el ángulo entre los vectores como distancia.

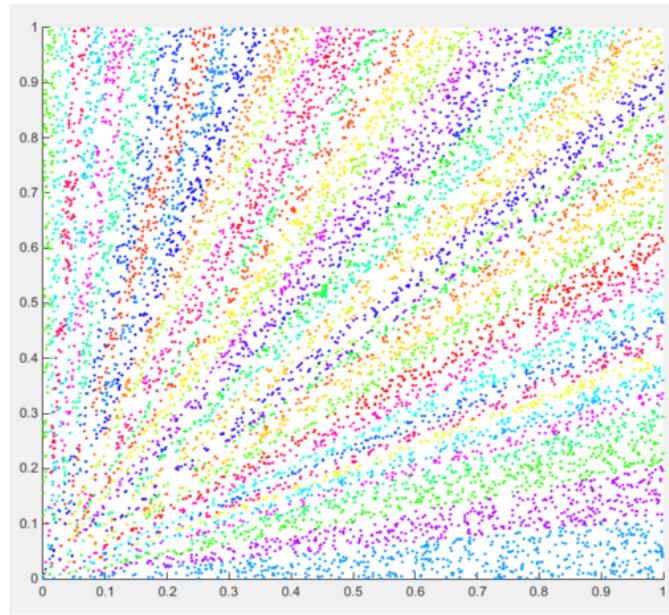


Figure 14.24: K-Means esférico

El resultado, además de ser muy bonito, muestra claramente como el algoritmo agrupa de acuerdo al ángulo entre los puntos. Ahora vamos a hacer el mismo experimento pero previamente hemos normalizado los puntos para que tengan norma=1.

Aquí vemos exactamente el mismo resultado que antes pero con todos los puntos a distancia 1 del origen por haber sido normalizados.

14.2.11 Kernel K-Means

Kernel K-Means es simplemente la aplicación de un kernel al algoritmo K-Means, de esta forma logramos clustering cuando los clusters no son linealmente separables. El funcionamiento y objetivo del kernel es el mismo que hemos usado con SVM: transformar los datos a un espacio de mayor cantidad de dimensiones, incluso infinitas dimensiones, de forma que sean linealmente separables. En este caso que los clusters queden separados linealmente que es un requisito para K-Means.

Para poder usar kernels en K-Means tenemos que reducir toda la formulación del algoritmo al uso de productos internos. En primer lugar vamos a ver que es

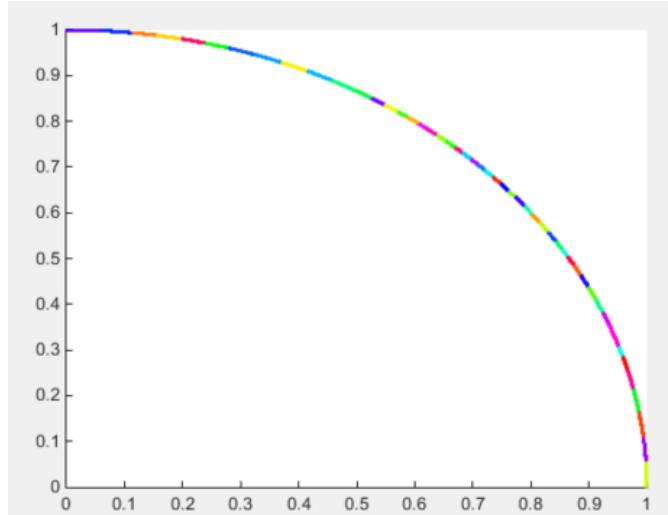


Figure 14.25: K-Means esférico

sencillo reducir la distancia entre dos puntos a productos internos:

$$|x - y|^2 = x^2 + y^2 - 2xy = \langle x, x \rangle + \langle y, y \rangle - 2\langle x, y \rangle \quad (14.14)$$

Con esta fórmula podemos usar cualquier kernel para calcular la distancia entre dos puntos, el problema en K-Means es el cálculo de los centroides. Los centroides se obtienen como un promedio de los puntos que pertenecen al cluster pero a su vez estos centroides residen en un espacio de muchas o infinitas dimensiones usando kernels, cosa que no podemos calcular. El truco es calcular directamente la distancia entre un punto y cada uno de los centroides sin tener que calcular los centroides. Veamos de qué forma esto es posible.

Planteamos la distancia entre un punto y un centroide:

$$\|\phi(x) - \frac{\sum \phi(a_i)}{N_i}\|^2 \quad (14.15)$$

Phi es la función que nos proyecta a un espacio de dimensiones superiores y que no sabemos ni queremos calcular. Nies la cantidad de puntos en el cluster ‘i’

Desarrollamos ahora la fórmula expandiendo el binomio:

$$\phi(x) * \phi(y) + \frac{\sum_{a_j, a_l \in i} \phi(a_j)\phi(a_k)}{N_i^2} - 2\frac{\sum_{a_j \in i} \phi(a_j)\phi(x)}{N_i} \quad (14.16)$$

Y ahora tenemos todo expresado en productos internos y podemos reemplazar por los kernels

$$\|x - c_i\|^2 = K(x, x) + \frac{\sum_{a_j, a_l \in i} K(a_l, a_j)}{N_i^2} - 2 \frac{\sum_{a_j \in i} K(a_j, x)}{N_i} \quad (14.17)$$

La primera sumatoria implica el kernel entre todos los puntos que pertenecen al cluster. La segunda sumatoria usa el kernel entre el punto “x” que queremos asignar y todos los que pertenecen al cluster “i”.

Notemos que la forma en que funciona K-Means ahora se ha modificado ya que se hace todo en un solo paso.

En primer lugar asignamos a cada punto un cluster al azar entre 1 y k.

Luego por cada punto calculamos la distancia entre el punto y cada cluster con la fórmula anterior y asignamos el punto al cluster más cercano.

Este proceso se repite hasta lograr la convergencia o bien hasta superar un cierto número de iteraciones. Notemos que al asignar el punto al cluster más cercano estamos modificando las asignaciones que usamos para calcular la distancia misma por lo que solo hace falta este paso para lograr la convergencia. En otras palabras la asignación de los puntos a los centroides cambia los centroides sin necesidad de tener que calcular estos últimos.

Para que Kernel - K-Means sea eficiente es necesario pre-calcular la matriz de kernel entre todos los puntos por lo que es necesario que la cantidad de puntos en total no sea muy grande. No es un algoritmo eficiente para datos masivos.

14.2.12 K-Modes

K-Modes es una variante de K-Means para datos que tienen atributos categóricos. Es posible usar K-Means con atributos categóricos convirtiéndolos usando One-Hot encoding o métodos similares que ya hemos visto. K-Modes tiene como ventaja que no se necesita conversión alguna de los datos.

El algoritmo en sí es idéntico al que ya conocemos, solo tenemos que redefinir de que forma hacemos el promedio de puntos para calcular los centroides y de que forma medimos las distancias entre puntos y centroides.

Para medir la distancia entre dos puntos vamos a usar la norma l_0 , es decir la cantidad de atributos en los cuáles los datos difieren. Por ejemplo si $x_1 = (3, \text{blue}, Y, \text{Chicago})$ y $x_2 = (5, \text{blue}, Y, \text{Denver})$ la distancia entre x_1 y x_2 es 2 ya que los puntos difieren en dos atributos.

Para calcular el “promedio” entre n puntos tomaremos por cada dimensión el valor que mas veces aparece, es decir la “moda” y por eso el nombre K-Modes. Si hay empate desempataremos al azar.

En general esto funciona bien cuando la cantidad de atributos es muy grande para que pueda haber mayor precisión en el rango de distancias posibles.

14.2.13 K-Means y el Problema de los Vecinos Mas Ceranos

Es posible usar K-Means para aproximar el problema de los vecinos mas cercanos (KNN) de forma eficiente. En primer lugar analicemos la posibilidad de correr K-Means sobre el set de datos con una cierta cantidad de centroides k y luego usar la estructura resultante para resolver el problema de los vecinos mas cercanos.

Si tenemos n puntos en d dimensiones y usamos $k = \sqrt{n}$ K-Means nos devuelve los k centroides y el centroide asignado a cada punto, es decir que cada uno de los n puntos queda asignado a un centroide entre 1 y k . Para buscar los puntos mas cercanos a un cierto punto q tenemos que comparar q contra cada uno de los k centroides y luego para el centroide mas cercano comparar contra todos los centroides asignados a dicho punto. Para esto podemos usar un índice invertido en donde tenemos k entradas y por cada entrada la lista de puntos que fueron asignados a dicho centroide.

Esto nos da un algoritmo que es $O(2\sqrt{m}) = O(\sqrt{m})$ esto es muy eficiente ya que para millones de puntos solo tenemos que realizar miles de comparaciones. El proceso puede hacerse online usando K-Means online e ir consultando a medida que se van agregando puntos con lo cual sirve también como un algoritmo de Streaming.

Veamos el resultado usando puntos al azar:

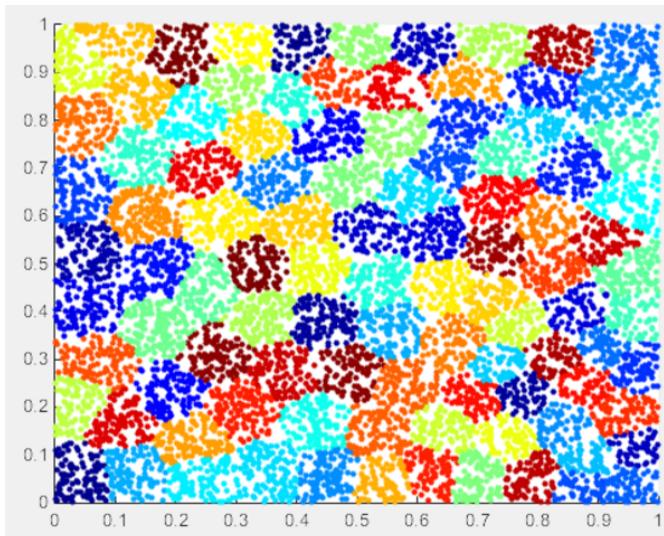


Figure 14.26: LSH via Kmeans

El resultado es un diagrama de Voronoi, como vemos cada uno de los clusters tiene aproximadamente 100 puntos. Cuando queremos hacer una consulta el

punto a consultar cae dentro de alguno de los 100 clusters existentes y lo comparamos contra los miembros de dicho cluster. Si el punto pertenece a la frontera del cluster entonces es posible que estemos dejando de lado algunos puntos que pueden ser similares. Más sobre esto más adelante.

Usando este método para encontrar imágenes similares en CIFAR-10 obtenemos este ejemplo:

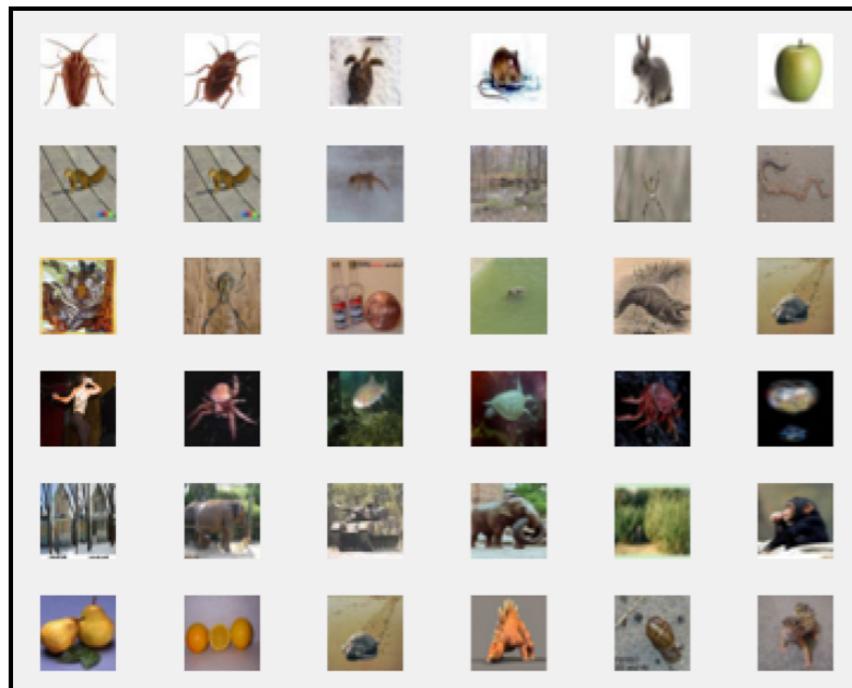


Figure 14.27: LSH via Kmeans para CIFAR-10

Los resultados son buenos y no es necesario ningún tipo de parametrización especial. Lo mas interesante es que el método escala muy bien incluso para millones de imágenes dado que el proceso de K-Means se hace una única vez al principio y hemos visto variantes muy eficientes de K-Means. Por ejemplo puede usarse K-Means online que corra en background continuamente actualizando los centroides en base a las imágenes para obtener cada vez mejores resultados.

En nuestro algoritmo a partir de una consulta es necesario en primer lugar comparar la misma contra cada uno de los centroides para obtener el más cercano, por lo tanto podemos aprovechar y usar las distancias a los centroides para ver si no hay dos centroides que estén muy cercanos al punto que buscamos en cuyo caso podemos buscar en ambos. Es una negociación de precisión por performance.

Datos Masivos, realmente masivos

Si los puntos son tantos que la cantidad de puntos a buscar por cada centroide se hace muy grande una posible solución es usar un valor de k mas grande para tener menor cantidad de puntos asignados a cada centroide pero es evidente que esta solución no es eficiente porque tenemos que comparar el query contra una cantidad mayor de centroides y además el proceso de K-Means se vuelve mucho mas lento. En el extremo si $k = n$ volvemos a la búsqueda por fuerza bruta. Es decir que aumentar el valor de k solo nos acerca a la búsqueda por fuerza bruta que es precisamente lo que queremos evitar.

Vamos a ver dos soluciones para este problema: una basada en una estructura llamada *inverted multi index* y otra usando árboles basados en K-means, ambas soluciones solo tienen sentido si tenemos cientos de millones o incluso billones de puntos a los cuales queremos aplicar KNN.

14.2.14 The Inverted Multi-Index

En este algoritmo [Babenko-Lempitsky] vamos a empezar dividiendo las columnas d del set de datos en dos bloques de $d/2$ columnas cada uno. Luego vamos a usar K-Means sobre cada uno de estos bloques como si fuera un set de datos independiente, es decir primero corremos K-Means con las primeras $d/2$ columnas de los datos y luego con las últimas $d/2$ columnas. Esto nos da dos juegos de centroides independientes U y V y dos centroides asignados a cada punto. Ahora en lugar de tener un índice invertido por cada centroide tendremos un índice por cada *par* de centroides.

Al usar dos valores la cantidad de puntos en cada *celda* del índice se reduce notablemente, es posible incluso que en una celda no tengamos la cantidad de puntos necesarios para encontrar los k vecinos mas cercanos a un cierto query.

Para el query, simplemente buscaremos los dos centroides mas cercanos al mismo y esto nos dará la primera entrada del índice invertido múltiple a buscar. Por ejemplo (2,3) indicando que el query es cercano al cluster 2 del primer K-means y al cluster 3 del segundo K-means. Al ir comparando el query con los $2k$ centroides podemos calcular la distancia del mismo a cada centroide por lo que tendremos para cada combinación de centroides una distancia que es la suma de la distancia del query a cada centroide. Para resolver el problema de KNN vamos a acceder al índice invertido en orden de distancias decreciente hasta completar los k puntos.

Hay múltiples variantes de esta estructura, por ejemplo se puede aplicar primero la SVD al set de datos sin reducir la cantidad de dimensiones, luego correr los 2 K-Means como hemos descripto y luego dentro de cada celda podemos volver a aplicar el algoritmo como si cada celda fuera un set de datos independiente. Esta estructura recursiva da lugar a varios algoritmos que son extremadamente eficientes como por ejemplo LOPQ [Kalantidis].

Estos métodos son muy necesarios, por ejemplo para búsqueda de imágenes similares a otras en donde podemos tener billones de puntos (imágenes) en millones de dimensiones. Para estos sets de datos KNN suele ser el mejor algoritmo

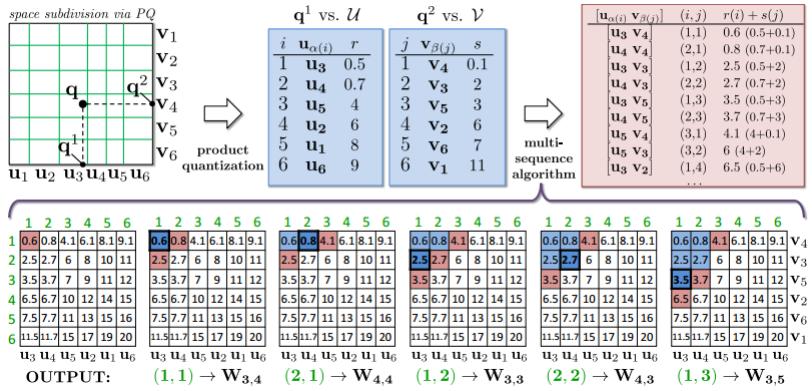


Figure 14.28: Inverted Multi Index: En primer lugar calculamos la distancia desde el query Q a cada centroide usando $d/2$ columnas de Q en cada caso. Luego en base a las distancias entre el query y cada centroide calculamos la suma de las dos distancias (r y s). Por lo tanto vamos a acceder primero al índice por las posiciones (3,4) cuya distancia sumada es 0.6, luego vamos a (4,4) con distancia 0.8, luego a (3,3) y así sucesivamente. Cada uno de estos pares sirve como clave de un índice invertido que almacena que puntos del set de datos original quedaron asignados a cada par de centroides

para encontrar semejanzas, en algunos casos extrayendo features de las imágenes usando algoritmos como por ejemplo SIFT. Incluso reduciendo la cantidad de dimensiones a cientos o miles KNN es completamente inviable y son necesarias aproximaciones. Los Multi-Inverted indexes, LOPQ o K-Means Trees son una solución a este problema y por eso su importancia.

Hablando de estructuras recursivas una idea simple es aplicar K-Means al resultado de K-Means...

K-means Trees

Para datos que son realmente masivos o streams el uso de la versión online de K-Means es necesario pero aún así podemos tener el problema de que la cantidad de comparaciones en cada paso sea muy costosa, por ejemplo con 1 millón de datos necesitamos 1000 comparaciones por consulta y puede que este sea un número demasiado alto.

Es posible acotar la cantidad de comparaciones que hacemos en cada punto construyendo múltiples niveles de K-Means. A esto lo llamamos “árboles de K-Means” cada nivel tiene una cantidad fija de centroides y una cantidad fija de puntos. Cuando la cantidad de puntos en una de las ramas del árbol supera el máximo creamos un nuevo nivel de K-Means en donde cada punto va a ser un centroide y no tendrá puntos asociados, de esta forma podemos ir llenando

el árbol garantizando tiempo constante para inserciones y consultas.

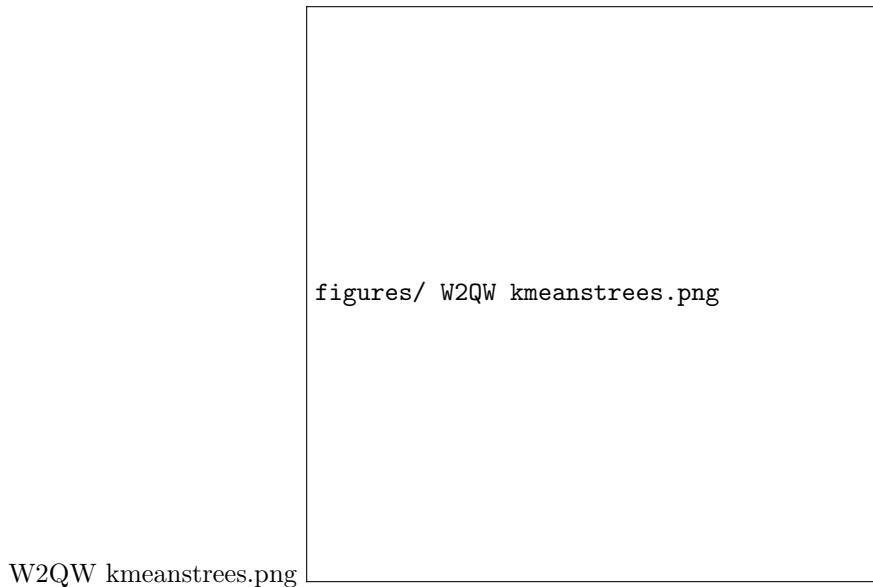


Figure 14.29: Ejemplo de K-Means recursivo en dos niveles. En primer lugar se hace un K-Means con $k = 9$, luego un K-Means de estos 9 centroides con $k = 3$. Si nuestro query es el punto rojo tenemos que comparar primero con los tres centroides fucsias ir al mas cercano y repetir el proceso, usando backtracking si es necesario hasta recolectar los k vecinos mas cercanos al query.

El éxito de este algoritmo comparado contra LOPQ o los inverted multi indexes depende en gran medida del set de datos y de la forma en la cual los datos se distribuyen en el mismo.

14.3 Clustering Espectral

El algoritmo de Clustering Espectral [Weiss-Ng-Jordan] surge originalmente como una forma de realizar cortes entre componentes de un grafo, siendo un algoritmo originalmente pensado para grafos pasó un cierto tiempo hasta que se comenzó a usar para cualquier tipo de datos. Este algoritmo, al igual que Kernel K-Means, tiene la capacidad de detectar formas de clusters arbitrarias, no-lineales.

Sea $X_{m \times n}$ nuestra matriz de m datos en n dimensiones. Para transformarla en un grafo vamos a usar el mismo proceso que hemos usado en Laplacian Eigenmaps por lo que solamente lo repasaremos muy rápidamente.

14.3.1 Primer paso: construir la matriz de afinidad

$$W_{ij} = \exp - \frac{\|x_i - x_j\|^2}{2\sigma^2} \quad (14.18)$$

Siendo σ un parámetro que queremos determinar.

14.3.2 Segundo paso: construir la matriz Laplaciana

“En un grafo G la matriz Laplaciana del mismo contiene la información sobre la conectividad de los diferentes componentes del grafo”

La matriz Laplaciana tiene varias construcciones posibles pero la más común es:

$$L = D - W \quad (14.19)$$

Siendo D la matriz diagonal con el grado de cada vértice del grafo, donde por grado debemos entender la suma de cada columna de W_{ij} . Cuando “W” es la matriz de adyacencia de un grafo el grado es la cantidad de vértices conectados al mismo. En nuestro caso al ser el grafo completo y con pesos en las aristas el “grado” será la suma de todos los pesos de las aristas que inciden sobre cada nodo. Notemos que trabajamos con grafos no-dirigidos.

Otras variantes para L:

$$L2 = D^{1/2}(D - W)D^{1/2} \quad (14.20)$$

$$L3 = D^{-1} - W \quad (14.21)$$

Propiedades oscuras, insondables de la teoría de grafos nos dicen que el espectro de la matriz Laplaciana sirve para encontrar cortes mínimos en un grafo. Un corte mínimo es aquel que divide el grafo en dos conjuntos de nodos de forma tal que las aristas entre nodos de los dos conjuntos tienen afinidad mínima. Es decir que podemos usarlo para separar el grafo en componentes lo cual equivale a realizar clustering entre los puntos que hemos representado como grafos del nodo.

El párrafo anterior explica la esencia del clustering espectral, si lo pasaron de largo por favor vuelvan a leerlo!

Una propiedad interesante de los autovalores de la matriz Laplaciana de un grafo es que la cantidad de autovalores iguales a cero es igual a la cantidad de componentes conexos en el grafo, recordemos que la matriz Laplaciana representa la conectividad del mismo. En un grafo completo tendremos un solo autovalor igual a cero, en un grafo sin aristas tendremos que todos los autovalores son cero lo cual tiene mucho sentido.

14.3.3 Tercer paso: Autovalores y Autovectores de la matriz Laplaciana

El siguiente paso es descomponer la matriz Laplaciana en sus autovalores y autovectores y determinar la cantidad de autovectores a usar. Esto lo podemos hacer analizando el plot de los autovalores de la matriz.

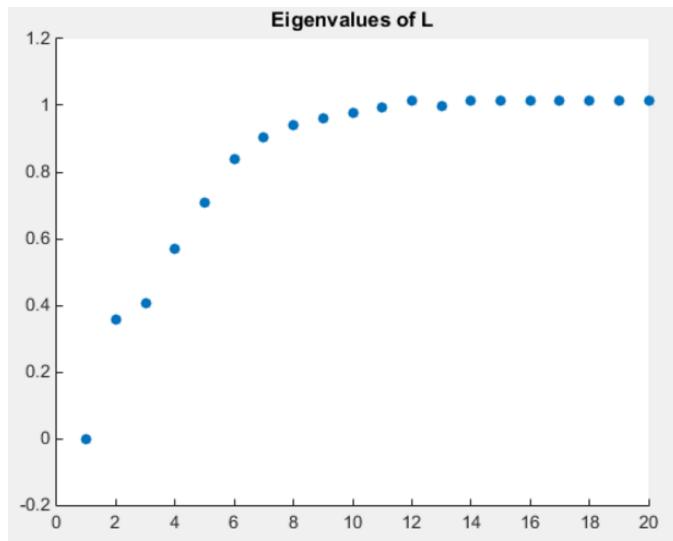


Figure 14.30: Autovalores de L

Tenemos un único autovalor en cero lo cual indica que el grafo es conexo. Lo que buscamos es un "salto" en los valores de los autovalores. Podemos ver en este caso que el salto se produce luego del tercer autovalor. Como el autovalor 0 lo descartamos podríamos deducir que en este caso en particular es conveniente usar 1 o 2 autovectores. Es decir que vamos a representar a nuestros datos en 1 o 2 dimensiones.

Vamos a volver pronto a los autovalores, grafiquemos ahora los primeros autovectores, es decir los asociados a los autovalores más pequeños.

Ordenamos los autovectores en el mismo orden que los autovalores:

Observando con detenimiento los autovectores podemos ver que tienen una estructura muy particular. Es decir toman valores muy similares para un cierto conjunto de puntos. Los autovectores no están mostrando ni más ni menos que la estructura de nuestra matriz de afinidad en diferentes niveles.

Es posible que uno solo de estos autovectores sea suficiente para realizar el clustering de nuestros datos, en general el primer autovector. Otras veces necesitamos un conjunto de autovectores. El algoritmo de clustering espectral es entonces el siguiente:

1. determinar el número de autovectores a usar de la matriz Laplaciana del

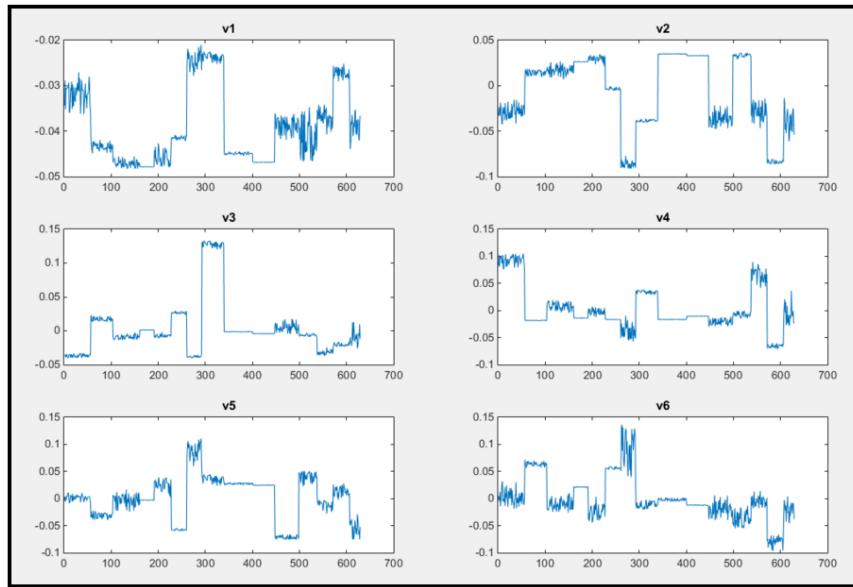


Figure 14.31: Autovectores de L

grafo

2. correr K-Means sobre estos autovectores
3. Asignar a los puntos originales los clusters de acuerdo al resultado de K-Means.

Algorithm 31: Clustering Espectral

- 1 Construir matriz W
 - 2 Construir matriz L
 - 3 Obtener los autovalores y autovectores de L
 - 4 Determinar z=el numero de autovalores a usar
 - 5 Aplicar K-Means a los z autovectores de L
 - 6 Asignar los puntos originales de acuerdo a las asignaciones de Kmeans
-

Observemos el resultado luego de aplicar K-Means en el espacio del espectro de la matriz Laplaciana:

El método de clustering espectral permite encontrar clusters de formas realmente arbitrarias como lo muestra el ejemplo:

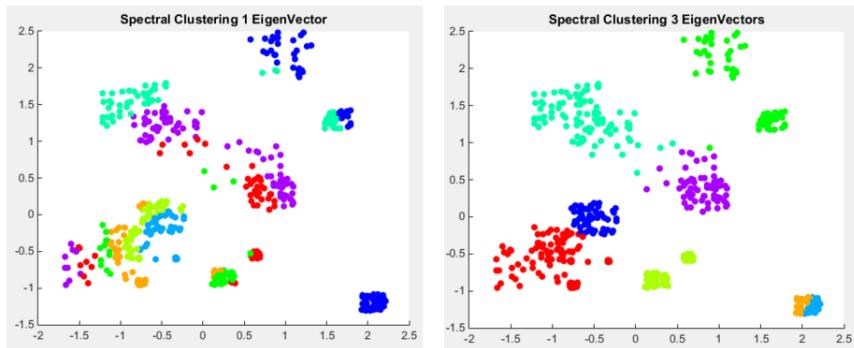


Figure 14.32: Clustering Espectral

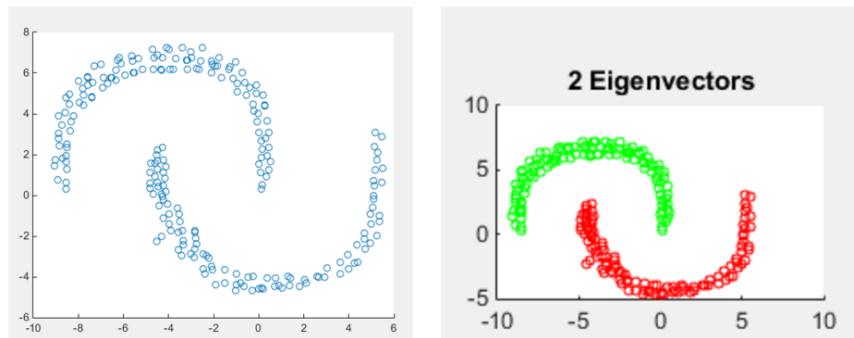


Figure 14.33: Clustering Espectral

14.3.4 Selección de Hiper-Parámetros

Al realizar clustering espectral hay que tener mucho cuidado en la selección de parámetros. Las cosas que hay que seleccionar son:

- El parámetro sigma para la matriz de afinidad
- La fórmula a usar para la matriz Laplaciana
- La cantidad de autovectores a usar

En muchas ocasiones esto implica un proceso complejo con varias pruebas antes de llegar a los resultados esperados. También implica que usar Spectral Clustering sin seleccionar los parámetros no tiene mucho sentido y puede producir cualquier tipo de resultado. En muchos casos la selección de hiperparámetros se puede automatizar usando la función costo de K-Means, aplicándola sobre los puntos originales y los centroides que surgirían del promedio de

los puntos que pertenecen a cada uno. Es decir que Clustering Espectral solo nos dice a que cluster pertenece cada punto y con esto podemos calcular los centroides y la distancia de cada punto a su centroide.

El algoritmo de clustering espectral es una herramienta muy poderosa ya que puede solucionar casos en los cuales la mayoría de los algoritmos falla, especialmente los casos en que K-Means no funciona en ninguna de sus innumerables variantes. El mecanismo por el cual solucionamos el problema es realizando el clustering en un espacio de dimensiones completamente diferente pero que surge en función del espacio original. La selección de los hiper-parámetros es crítica en el algoritmo de Clustering Espectral. Por más poderoso que sea el misil de nada sirve si no apunta a donde queremos.

14.3.5 Aproximaciones: KASP

El algoritmo de Clustering Espectral tiene como problema la necesidad de contar con la matriz de afinidad “W” que es una matriz de $m \times m$ siendo “m” la cantidad de datos. Esto puede llegar a ser un número imposible de manejar. Es necesario entonces realizar una aproximación que nos permita usar clustering espectral sin la necesidad de calcular W de forma completa. La aproximación que veremos se llama “Kasp” [Yan-Huan-Jordan] y está basada en K-Means.

El procedimiento es muy simple:

1. Usar K-Means para generar “z” centroides a partir de los datos.
2. Usar clustering espectral sobre los centroides.
3. Asignar a cada punto del set original el cluster al que pertenece su centroide.

Como vemos este procedimiento es realmente muy sencillo. Solo tenemos que usar primero K-Means, luego clustering espectral y simplemente recuperar para cada punto el centroide asignado por el algoritmo de clustering espectral que corresponde al centroide asignado por K-Means para el punto. Es notable que usamos Kmeans como método de compresión del set de datos de m a k puntos en donde k puede ser arbitrario.

Clustering Espectral puede aplicarse de forma directa a la detección de comunidades en grafos, por ejemplo en redes sociales, en este caso la matriz W es la matriz de adyacencia del grafo y el resto del algoritmo no requiere ningún cambio.

14.4 CURE

CURE [Guha-Rastogi-Shim] (Cluserint Using Representatives) es un algoritmo de clustering para grandes cantidades de datos, que es robusto a datos anómalos y permite clusters de forma arbitraria. CURE combina clustering jerárquico y K-Means de forma eficiente.

El primer paso es elegir un subconjunto de puntos al azar y agruparlos en k clusters usando clustering jerárquico. Es necesario conocer la cantidad de clusters que queremos k igual que en K-Means. Estos puntos no pueden ser muchos ya que, como sabemos, sería ineficiente realizar el proceso de clustering jerárquico.

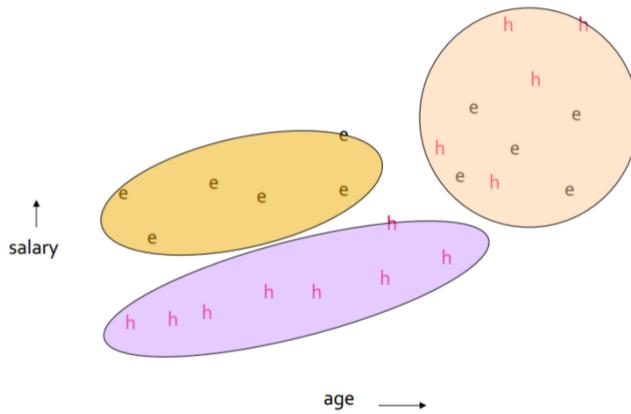


Figure 14.34: CURE 1

El segundo paso es de cada cluster formado elegir f puntos representativos que son f los puntos del cluster que están mas alejados entre sí. El tercer paso es mover los puntos representativos un 20% hacia el centro del cluster.

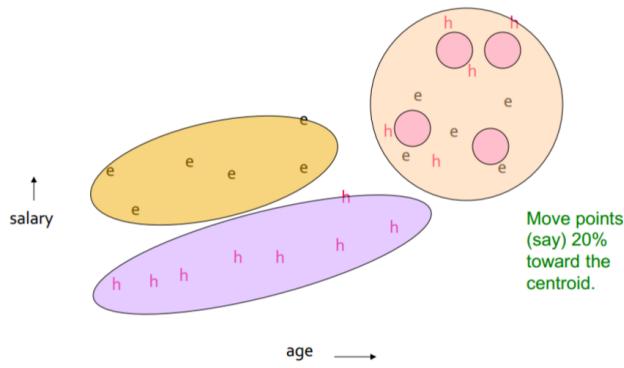


Figure 14.35: CURE 2

En el paso final por cada punto del set de datos que falta procesar lo asig-

namos al cluster que corresponde al punto representativo mas cercano al punto.

Algorithm 32: CURE

```

1 Elegir  $L$  puntos al azar
2 Aplicar clustering jerarquico para encontrar  $k$  clusters con dichos puntos
3 for  $i=1..k$  do
4   Seleccionar los  $f$  puntos mas representativos del cluster  $i$ 
5 while  $x$  not in  $L$  do
6   for  $i=1..k$  do
7     Comparar  $X$  con los puntos representativos en  $k$ 
8     Asignar  $x$  al cluster que corresponda al punto representativo mas
         cercano

```

El orden de CURE es $O(n^2 \log n)$ por lo que puede resultar ineficiente para datos masivos en cuyo caso hay que recurrir a algoritmos online.

14.5 DBScan

DBScan es un algoritmo de clustering basado en el concepto de "densidad" es decir encontrar zonas en las cuales haya una densidad de puntos altas e identificar estas zonas como un cluster. Gracias a esto DBscan es capaz de determinar automáticamente la cantidad de clusters y puede también detectar outliers, es decir puntos anómalos que no pertenecen a ningún cluster.

DBScan tiene dos hiper-parámetros: la distancia mínima entre dos puntos para estar en un mismo cluster que llamaremos " e " y la cantidad mínima de puntos vecinos a la distancia correcta para poder generar nuevos clusters " k ". Ambos son fáciles de interpretar, e nos dice la densidad con la cual vamos a agrupar puntos y k nos dice la cantidad mínima de puntos que puede tener un cluster. Los puntos que no pertenecen a un cluster y no son suficientes para formar un nuevo cluster son los outliers.

El algoritmo comienza con un punto cualquiera, si el punto no pertenece a un cluster entonces analizamos si tiene al menos $k - 1$ vecinos a distancia menor o igual a e en cuyo caso formamos un nuevo cluster con estos k puntos. Si el punto ya pertenece a un cluster entonces buscamos los puntos a distancia e o menor y los agregamos al cluster.

DBscan es capaz de detectar formas arbitrarias de casi cualquier tipo como por ejemplo:

Esto funciona siempre y cuando todos los clusters tengan densidades parecidas, es decir que los puntos dentro de cada cluster se encuentren a distancias similares. Cuando los clusters tienen distintas densidades DBscan falla ya que puede no ser posible encontrar un parámetro e que sirva.

En cuanto a performance tenemos el problema de po cada punto encontrar los puntos a distancia menor o igual a e que es $O(n^2)$ por fuerza bruta. La

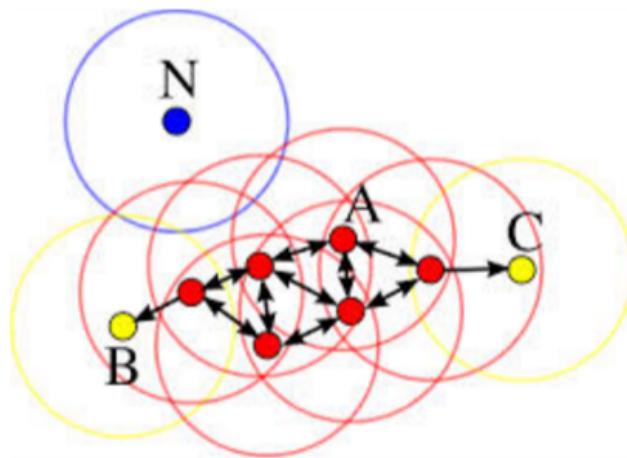


Figure 14.36: DBScan

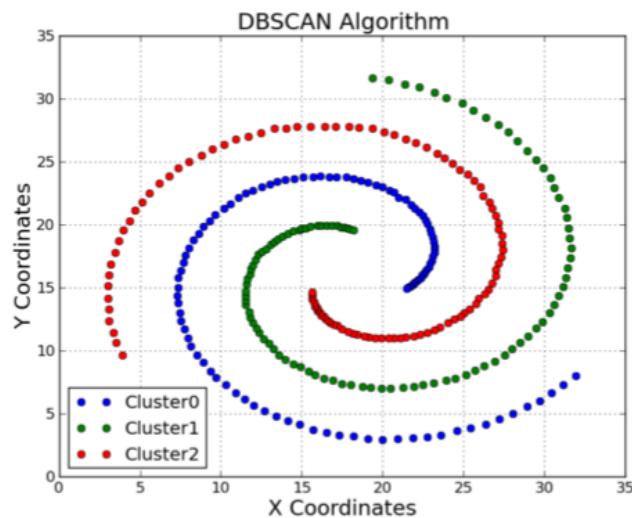


Figure 14.37: Ejemplo DBScan

solución en DBscan es usar algún tipo de índice espacial para llevar esto a $O(n \log n)$. Como sabemos estos índices no escalan bien en muchas dimensiones por lo que en espacios de muchas dimensiones DBScan sufre el problema de la

dimensión de la dimensionalidad igual que KNN.

14.6 HDBScan

HDBScan [Campello-Moulavi-Sander] es un algoritmo de clustering basado en DBScan que soluciona el problema de las densidades diferentes en los clusters. El problema es que si se usa una distancia muy pequeña ϵ los clusters con densidades de puntos bajas quedan sin identificar es decir como un enorme conjunto de outliers mientras que si usamos una densidad grande suficiente para encontrar los clusters menos densos empiezan a mergearse entre sí.

Vamos a usar a lo largo de esta sección el siguiente conjunto de puntos:

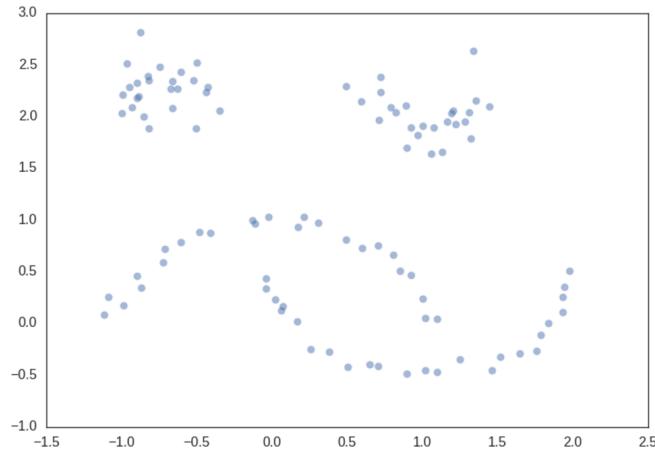


Figure 14.38: HDBScan ejemplo

El primer paso de HDBScan es construir una matriz de distancias entre los puntos pero en lugar de usar la distancia euclídea en HDScan se usa una distancia especial denominada "Mutual Reachability Distance (MRD)".

14.6.1 La Distancia MRD

Vamos a asociar a cada punto con la distancia de su vecindario que es la distancia máxima entre sus k vecinos más cercanos. k es un hiper-parámetro para el algoritmo.

A esta distancia la vamos a llamar el core_k del punto. Cada punto tiene asociada una cierta distancia correspondiente a $\text{core}_k(x_i)$ que, recordemos, es la distancia máxima entre el punto y sus k vecinos más cercanos.

La distancia MRD [Eldridge-Belkin] se define de la siguiente forma:

$$\text{MRD}(a, b) = \max\{\text{core}_k(a), \text{core}_k(b), d(a, b)\} \quad (14.22)$$

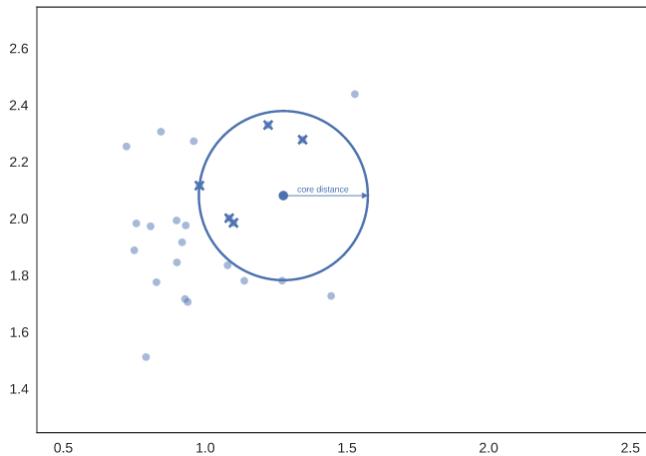


Figure 14.39: Vecindario de un punto

MRD funciona particularmente bien junto con clustering jerárquico que es uno de los algoritmos que va a usar HDBScan.

14.6.2 El árbol Generador Mínimo

Usando la matriz de distancias MRD el objetivo de HDBScan es encontrar las "islas" de puntos que tienen alta densidad en donde la densidad es variable para cada isla. Conceptualmente lo que vamos a hacer es considerar a los datos como un grafo en donde el peso de las aristas es la distancia MRD entre los puntos.

Pensemos ahora que podríamos ir variando un cierto umbral y deshacernos de las aristas que tengan una distancia mayor al mismo, eventualmente tenemos una jerarquía de componentes conexos que descubre las "islas" del grafo que estábamos buscando.

Una forma análoga de hacer esto es construyendo el árbol generador mínimo para el grafo. Esto se puede hacer de forma muy eficiente mediante el algoritmo de Kruskal. Empezamos con un grafo vacío y en cada paso agregamos la arista de menor peso que conecte el árbol actual a un nodo que todavía no forme parte del árbol. Notemos que el árbol generador usando la distancia MRD no es igual al árbol generador usando la distancia euclídea.

Una vez que tenemos el árbol generador mínimo vamos a usar el algoritmo de clustering jerárquico sobre estos puntos usando la distancia MRD.

14.6.3 Clustering Jerárquico

El resultado es un dendrograma, en este momento necesitamos usar la información en el dendrograma para identificar cuales son los clusters en nuestro set de datos. De abajo hacia arriba el dendrograma puede interpretarse como que

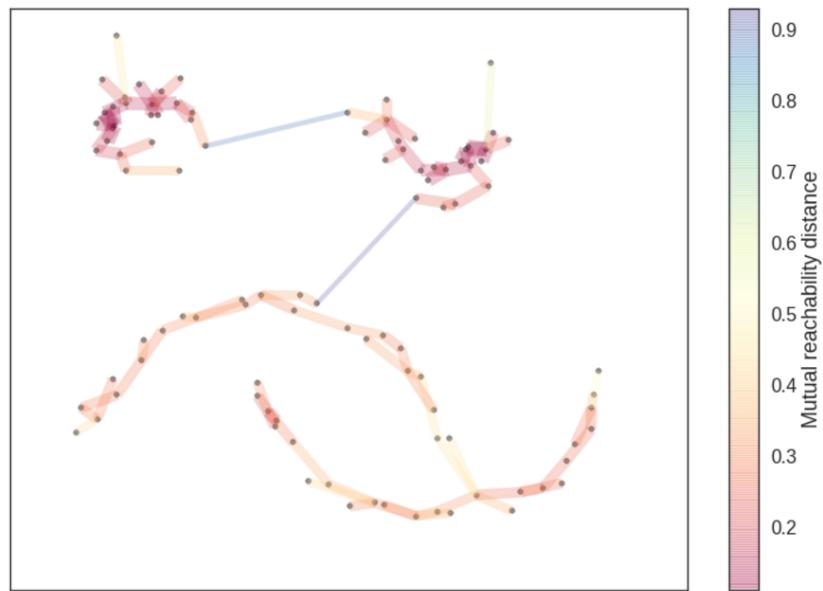


Figure 14.40: árbol generador mínimo

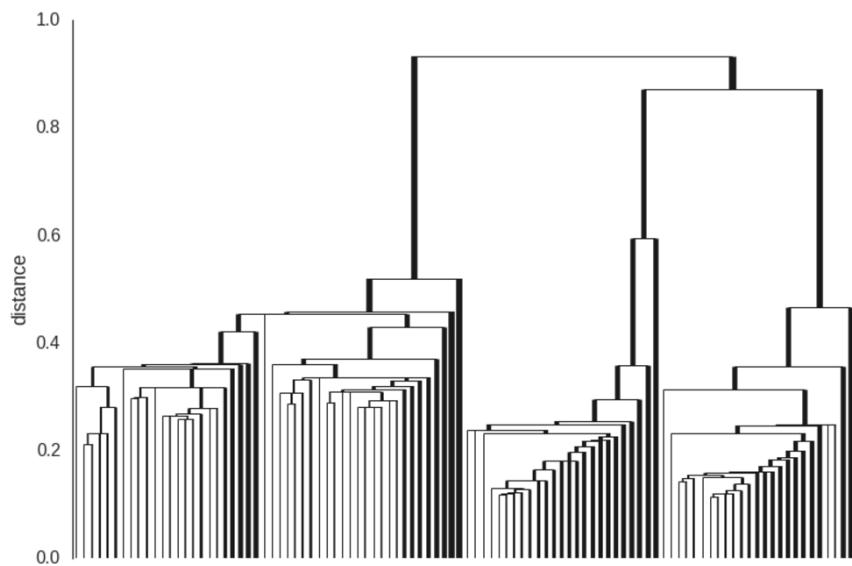


Figure 14.41: Clustering Jerárquico sobre los puntos del árbol generador mínimo

clusters fuimos uniendo para formar clusters mas grandes, en este caso eso no nos interesa sino que queremos recorrer el dendrograma de arriba hacia abajo. Si recorremos el dendrograma de arriba hacia abajo partimos de un único cluster con todos los puntos que se va subdividiendo en otros clusters, al igual que en DBScan nos interesa fijar un tamaño mínimo para que un conjunto de puntos se pueda considerar un cluster. Para que un split de nuestro dendrograma genere dos nuevos clusters éstos tienen que tener al menos la cantidad mínima de puntos que establecimos, en caso contrario lo vamos a considerar como un cluster que pierde puntos pero sigue siendo el mismo cluster.

Esto nos permite podar el dendrograma en donde solo quedan los splits que generan clusters válidos. La cantidad de puntos en los clusters puede ir variando porque los clusters que no se subdividen pueden perder puntos, esto lo podemos representar con un nuevo dendrograma de la siguiente forma:

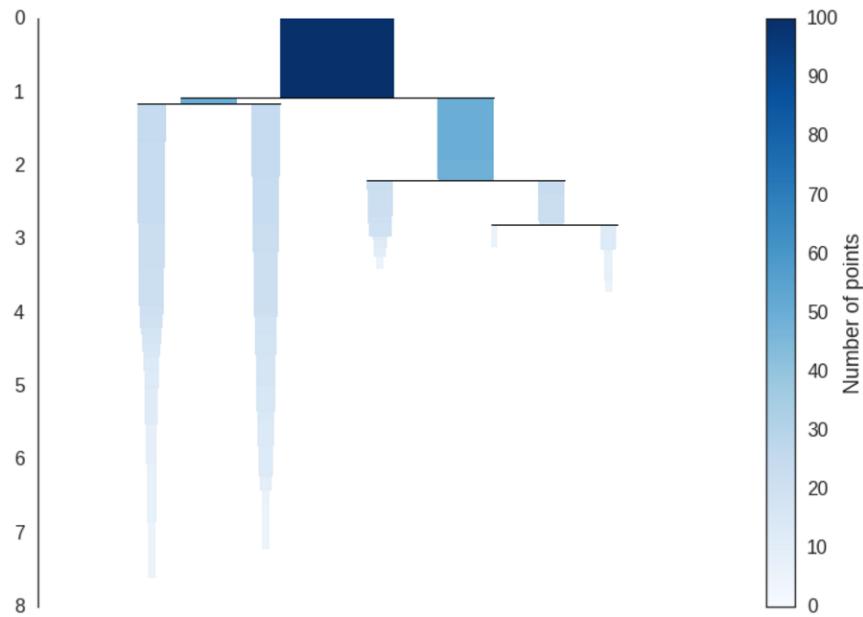


Figure 14.42: Dendrograma luego de la poda

14.6.4 Extraer los Clusters

Empezamos con un único cluster con muchos puntos, este se subdivide en 2, la rama izquierda inmediatamente se divide en dos clusters que perduran en el tiempo perdiendo algunos puntos. La rama derecha se dividen en el paso 2 en dos clusters el de la rama izquierda queda y el de la rama derecha se divide en 2 en el paso 3. Esta representación simplificada tiene información muy importante

sobre los clusters en nuestro set de datos.

Intuitivamente queremos aquellos clusters que persisten a lo largo del tiempo, los clusters que duran muy poco probablemente sean simplemente ruido. Observando el dendrograma luego de la poda podemos decir que queremos los clusters que tienen mayor área en el plot. Si seleccionamos un cluster no podemos luego elegir a ningún descendiente del mismo, esto es obvio porque compartirían puntos.

En primer lugar necesitamos una medida de distancia para medir la persistencia de los clusters. Recordemos que en cada paso del clustering jerárquico tenemos la distancia a la que estaban los clusters unidos, vamos a usar $\lambda = 1/distance$. Por cada cluster tenemos entonces dos lambdas, uno que es el valor con el cual el cluster nace y otro con el cual el cluster muere (es decir se subdivide en dos clusters de al menos p puntos). Llamemos a estos λ_{birth} y λ_{death}

Además, por cada punto p en un cluster tenemos λ_p que es el valor de lambda en el momento en el cual el punto "cayó" fuera del cluster, este valor va a estar entre λ_{birth} y λ_{death} ya que el punto o bien cae fuera durante la vida del cluster o bien abandona el cluster cuando el cluster muere. Notemos que el hecho de que un punto "caiga" fuera de un cluster no quiere decir que el punto deje de pertenecer al cluster.

Vamos a definir la *estabilidad* de un cluster como:

$$\sum_{p \in \text{cluster}} (\lambda_p - \lambda_{\text{birth}}) \quad (14.23)$$

Tenemos ahora todos los elementos para explicar el algoritmo con el cual seleccionaremos los clusters.

Empezamos con cada hoja del dendrograma podado como un cluster seleccionado. Vamos a recorrer el árbol hacia arriba (ordenamiento topológico inverso). Si la suma de la estabilidad de los dos hijos es mayor que la estabilidad del padre entonces seteamos la estabilidad del padre como la suma de la estabilidad de sus hijos. Si la suma de la estabilidad de los hijos es menor que la estabilidad del padre entonces declaramos al cluster padre como un cluster seleccionado y des-seleccionamos a los hijos. Una vez que llegamos a la raíz el conjunto de clusters seleccionados constituye nuestro grupo de clusters.

Una vez que tenemos los clusters es fácil llegar al resultado final. Cualquier punto que no esté en alguno de los clusters seleccionados es ruido y le asignamos un label -1. Por cada cluster tenemos el valor de λ_p para cada punto del cluster, si normalizamos estos valores en el rango 0..1 tenemos una medida de pertenencia de cada punto al cluster. Esto lo podemos usar como la probabilidad de que cada punto pertenezca al mismo. La solución final para nuestro ejemplo es:

La intensidad del color refleja la probabilidad de que el punto pertenezca al cluster. En este caso no tenemos outliers, todos los puntos quedaron identificados dentro de algún cluster.

En el segundo ejemplo vemos como HDBScan identifica los clusters principales en el set de datos independientemente de su forma o densidad y como quedan como outliers los puntos para los cuales no hay una forma lógica de

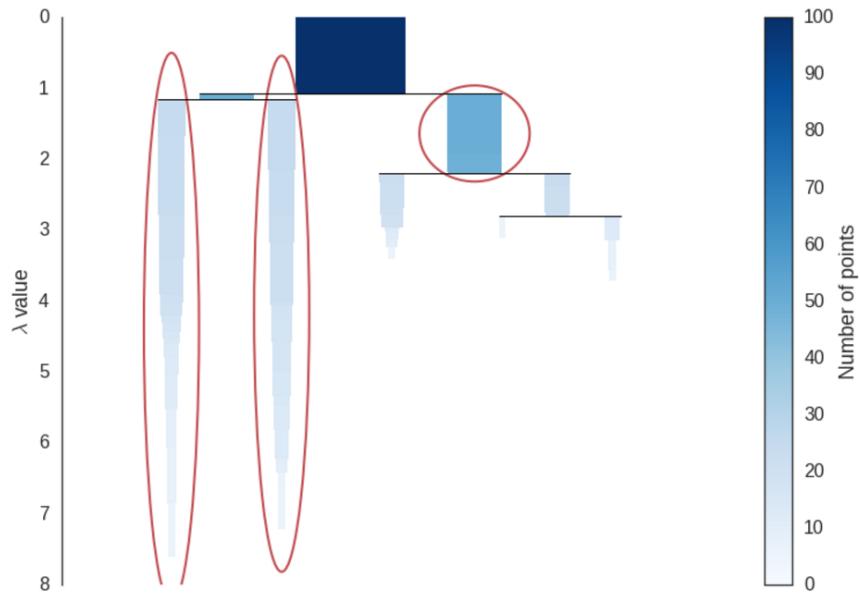


Figure 14.43: Seleccionando los Clusters

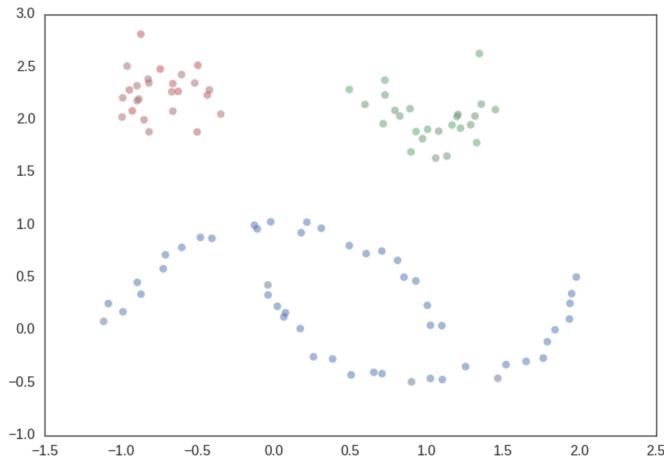


Figure 14.44: Resultado Final de HDBScan

asociarlos a un cluster. Un punto a considerar es que algunos puntos que, aparentemente, deberían quedar dentro de un cluster aparecen como outliers, HDBScan tiende a detectar mas outliers de los necesarios, esto puede corre-

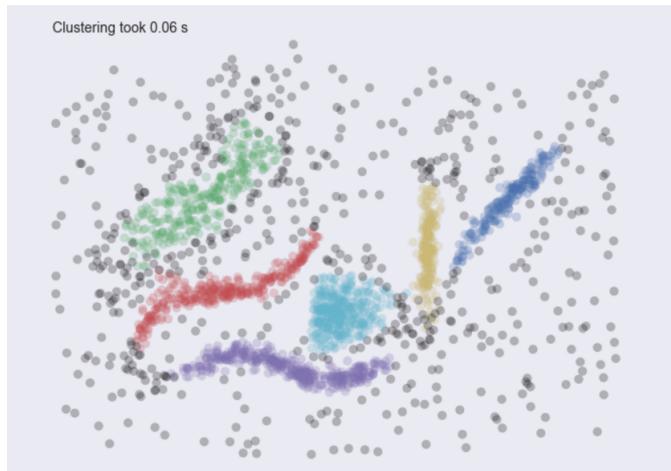


Figure 14.45: Otro ejemplo de HDBScan

girse controlando el porcentaje de outliers y variando los hiper-parámetros del algoritmo.

HDBScan es un algoritmo muy interesante ya que permite encontrar clusters de forma arbitraria y con densidades arbitrarias, identificar outliers y detectar automáticamente el número de clusters.

14.7 Bioclustering

Supongamos que tenemos un set de documentos y la matriz de términos por documentos correspondiente al mismo. Un algoritmo de clustering tradicional puede usarse tanto para agrupar documentos como para agrupar términos usando la matriz o su transpuesta según el caso. Lo interesante es agrupar en base a ambas cosas, es decir crear clusters que representen a un conjunto de términos y documentos. A esto se lo conoce como bi-clustering o co-clustering.

La utilidad de este algoritmo es permitir agrupar los documentos en clusters y al mismo tiempo obtener una agrupación de cuáles son los términos que están mas asociados a esos documentos, notemos que esto no es lo mismo que clusterizar los documentos y luego extraer los términos mas significativos de cada cluster.

Vamos a ver dos algoritmos para bi-clustering, uno basado en clustering espectral y el otro basado en LSI. En primer lugar porque son los dos algoritmos que suelen dar mejores resultados y en segundo lugar porque es hermoso re-usar los temas que ya vimos antes.

14.7.1 Co-clustering Espectral

El algoritmo de co-clustering espectral [Dhillon] se basa en el principio de aplicar clustering espectral al Laplaciano del grafo bipartito entre las filas y las columnas de una matriz. Afortunadamente el grafo que sería de $(m+n)x(m+n)$ no es necesario ya que puede derivarse de la matriz de datos que llamaremos A , en nuestro ejemplo esta es simplemente la matriz de términos x documentos. Por ejemplo:

$$A = \begin{pmatrix} 5 & 1 & 4 & 0 \\ 1 & 8 & 0 & 5 \\ 5 & 0 & 5 & 1 \\ 1 & 4 & 1 & 4 \end{pmatrix}$$

En primer lugar vamos a calcular una matriz diagonal R que tendrá la suma por filas de la matriz A y una matriz C que será una matriz diagonal con la suma por columnas de la matriz A , se llaman R y C por "rows" and "columns". Muy ingenioso.

$$R = \begin{pmatrix} 10 & 0 & 0 & 0 \\ 0 & 14 & 0 & 0 \\ 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 10 \end{pmatrix} C = \begin{pmatrix} 12 & 0 & 0 & 0 \\ 0 & 13 & 0 & 0 \\ 9 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{pmatrix} \quad (14.24)$$

El siguiente paso es obtener la matriz normalizada que llamaremos A_n que se obtiene mediante la fórmula:

$$A_n = R^{-1/2} A C^{-1/2} \quad (14.25)$$

En nuestro caso el resultado es:

$$A_n = \begin{pmatrix} 0.4564 & 0.0877 & 0.4000 & 0 \\ 0.0772 & 0.5930 & 0 & 0.4226 \\ 0.4352 & 0 & 0.4767 & 0.0953 \\ 0.0913 & 0.3508 & 0.1000 & 0.4000 \end{pmatrix}$$

La matriz A_n sirve como matriz Laplaciana del grafo bipartito entre las filas y columnas de A .

El siguiente paso es calcular la SVD de la matriz normalizada:

$$A_n = U \Sigma V \quad (14.26)$$

$$U = \begin{pmatrix} -0.4714 & 0.4879 & 0.5598 & 0.4757 \\ -0.5578 & -0.5989 & 0.4016 & -0.4111 \\ -0.4944 & 0.5357 & -0.4317 & -0.5313 \\ -0.4714 & -0.3411 & -0.5822 & 0.5678 \end{pmatrix}$$

$$\Sigma = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.7860 & 0 & 0 \\ 0 & 0 & 0.1465 & 0 \\ 0 & 0 & 0 & 0.0095 \end{pmatrix} \quad (14.27)$$

$$V = \begin{pmatrix} -0.5164 & 0.4815 & 0.3106 & 0.6364 \\ -0.5375 & -0.5496 & 0.5665 & -0.2968 \\ -0.4714 & 0.5298 & -0.2737 & -0.6498 \\ -0.4714 & -0.4306 & -0.7125 & 0.2910 \end{pmatrix}$$

Vamos a descartar la primer columna de U y V y quedarnos con las siguientes $\log_2(k)$ columnas en donde k es la cantidad de clusters que queremos formar. En nuestro ejemplo estamos interesados en formar 2 clusters por lo tanto $k = 2$ y nos vamos a quedar solo con la segunda columna de U y V , en general son varias las columnas a conservar y no solo una.

Vamos a normalizar las columnas de la siguiente forma:

$$U_2 = R^{-1/2}U[:, 2 : (2 + \log_2(k))] \quad (14.28)$$

$$V_2 = C^{-1/2}V[:, 2 : (2 + \log_2(k))] \quad (14.29)$$

Finalmente formamos la matriz Z concatenando (x filas) las matrices U_2 y V_2 nuestro resultado queda:

$$Z = \begin{pmatrix} 0.1543 \\ -0.1601 \\ 0.1615 \\ -0.1079 \\ 0.1523 \\ -0.1469 \\ 0.1597 \\ -0.1362 \end{pmatrix}$$

El último paso es aplicar K-Means sobre la matriz Z , las primeras m filas de Z nos dan la asignación de las filas de la matriz A a los clusters y las siguientes n filas nos dan la asignación de las columnas. Notemos que en nuestra matriz nos quedaron numeros positivos para las filas y columnas pares y negativos para las filas y columnas impares, lo cual tiene sentido para nuestra matriz A , por lo tanto luego de correr K-Means vamos a obtener un bi-cluster formado por $\{d1, d3, t1, t3\}$ y otro bi-cluster formado por $\{d2, d4, t2, t4\}$.

Notar que el algoritmo está usando la matriz U de la SVD para agrupar las filas y la matriz V de la SVD para agrupar las columnas, esto tiene mucho sentido si recordamos la interpretación algebraica de la SVD.

Algorithm 33: Spectral Co-Clustering

Data: A: document term matrix, k: number of clusters
Result: Documents and terms partitioned into clusters C

```

1 begin
2   R = diag(sum(A, 2));
3   C = diag(sum(A, 1));
4   An = R-1/2AC-1/2;
5   [U, Σ, V] = svd(An);
6   U2 = U[:, 2 : (2 + log2(k))];
7   V2 = V[:, 2 : (2 + log2(k))];
8   Z = [R-1/2U2; C-1/2V2];
9   C = Kmeans(Z)
10 end

```

El algoritmo de bi-clustering entonces es el siguiente:

14.7.2 LSI Co-clustering

El algoritmo basado en LSI es muy similar al anterior cambiando solamente el contenido de la matriz, es decir la forma en que normalizamos la misma. Antes suponíamos que teníamos una matriz de términos por documentos en donde indicábamos la cantidad de veces que cada término ocurría en cada documento. Ahora vamos a usar una matriz del estilo de la que usamos en LSI es decir basada en TF-IDF por ejemplo de la forma:

$$W_{ij} = \log_2(f_{tij}) \log\left(\frac{N}{f_{ti}} + 1\right) \quad (14.30)$$

Donde f_{tij} es la cantidad de veces que el término i ocurre en el documento j ; f_{ti} es la cantidad de documentos en los que ocurre el término i y N es la cantidad total de documentos en la colección.

A esta matriz que llamaremos L le aplicamos la SVD tal y como hacíamos en LSI. Una vez que hicimos esto tenemos que la matriz U nos da la relación entre los términos y los conceptos latentes y la matriz V nos da la relación entre los documentos y los conceptos latentes. A ambas matrices las vamos a multiplicar por la matriz de valores singulares Σ para darles el peso que corresponde. Luego de cada matriz tomamos $\log_2(k)$ columnas esta vez a partir de la columna 1.

Luego simplemente concatenamos los dos resultados por filas y aplicamos K-Means pero esta vez en lugar de aplicar K-Means tradicional tenemos que aplicar K-Means esférico es decir usando la distancia angular entre los vectores.

El algoritmo completo es:

Ambos algoritmos dan resultados similares [Park-Leckie] y son los algoritmos más usados para aplicar bi-clustering a una matriz de términos por documentos. Los resultados obtenidos son muy interesantes ya que permiten analizar los clusters en base a los términos que fueron clusterizados junto con los documentos.

Algorithm 34: LSI Co-Clustering

Data: A: TF-IDF matrix, k: number of clusters
Result: Documents and terms partitioned into clusters C

```

1 begin
2    $[U, \Sigma, V] = svd(A);$ 
3    $U_2 = U[:, 1 : (1 + \log_2(k))];$ 
4    $V_2 = V[:, 1 : (1 + \log_2(k))];$ 
5    $Z = [\Sigma U_2; V_2 \Sigma];$ 
6    $C = SphericalKmeans(Z)$ 
7 end

```

Por ejemplo luego de aplicar Spectral Co-Clustering a una serie de textos de newsgroups los resultados fueron los siguientes:

Best biclusters:

```

bicluster 0 : 1951 documents, 4373 words
categories   : 23% talk.politics.guns, 19% talk.politics.misc, 14% sci.med
words       : gun, guns, geb, banks, firearms, drugs, gordon, clinton, cdt, amendment

bicluster 1 : 1165 documents, 3304 words
categories   : 29% talk.politics.mideast, 26% soc.religion.christian, 25% alt.atheism
words       : god, jesus, christians, atheists, kent, sin, morality, belief, resurrect

bicluster 2 : 2219 documents, 2830 words
categories   : 18% comp.sys.mac.hardware, 16% comp.sys.ibm.pc.hardware, 16% comp.graphic
words       : voltage, dsp, board, receiver, circuit, shipping, packages, stereo, comp

bicluster 3 : 1860 documents, 2745 words
categories   : 26% rec.motorcycles, 23% rec.autos, 13% misc.forsale
words       : bike, car, dod, engine, motorcycle, ride, honda, cars, bmw, bikes

bicluster 4 : 12 documents, 155 words
categories   : 100% rec.sport.hockey
words       : scorer, unassisted, reichel, semak, sweeney, kovalenko, ricci, audette,

```

Se puede observar que los términos nos permiten interpretar el significado de cada uno de los clusters.

14.8 NMF: Non-negative matrix factorization

Este capítulo aborda el fascinante mundo de la factorización de matrices, concretamente las factorizaciones no-negativas. Vamos a ver que poder factorizar una matriz como el producto de dos matrices no-negativas tiene una enorme

cantidad de aplicaciones, algunas de ellas muy curiosas. Como buena noticia los algoritmos para resolver la factorización son realmente simples y elegantes y las aplicaciones son muy interesantes así que tendremos muy poco costo y un gran beneficio.

Antes de empezar con el tema debemos hacer dos aclaraciones importantes: En primer lugar vemos NMF dentro de Clustering pero es un tema que tiene muchas aplicaciones mas, simplemente había que ponerlo en algún lado. La segunda aclaración es que existen **muchos** algoritmos para NMF y solo podemos ver alguno de ellos.

Vamos a partir de una matriz A de $m \times n$ no-negativa. Queremos hallar dos matrices: W y H en donde W es una matriz de $m \times k$ no-negativa y H es una matriz de $k \times n$ no-negativa tales que:

$$A \approx W * H \quad (14.31)$$

Una forma alternativa de definir el problema es encontrar las matrices que minimicen la norma de Frobenius con la matriz original:

$$\min \frac{1}{2} \|A - WH\|^2 \quad (14.32)$$

14.8.1 Algoritmo Multiplicativo

Este algoritmo inicializa W y H al azar y luego actualiza las matrices mediante un algoritmo multiplicativo.

Algorithm 35: NMF Multiplicativo

```

1 inicializar  $W$  y  $H$  al azar
2 for  $i=1:maxiter$  do
3    $H = H .*(W^{(t)}A)./(W^{(t)}WH + 10^{-9})$ 
4    $W = W .*(AH^{(t)})./(WHH^{(t)} + 10^{-9})$ 

```

La suma de 10^{-9} es solo para evitar dividir por cero. Las operaciones $.*$ y $./$ implican el producto o división miembro a miembro de dos matrices.

Ejemplo:

$$X = \begin{pmatrix} 0.7431 & 0.7060 & 0.0971 & 0.9502 \\ 0.3922 & 0.0318 & 0.8235 & 0.0344 \\ 0.6555 & 0.2769 & 0.6948 & 0.4387 \\ 0.1712 & 0.0462 & 0.3171 & 0.3816 \end{pmatrix}$$

$$X \approx \begin{pmatrix} 1.5608 & 0.1184 \\ 0.0000 & 1.0588 \\ 0.6510 & 0.9086 \\ 0.3398 & 0.3549 \end{pmatrix} \begin{pmatrix} 0.4426 & 0.4362 & 0.0047 & 0.6197 \\ 0.3655 & 0.0000 & 0.7778 & 0.0610 \end{pmatrix} = \begin{pmatrix} 0.7341 & 0.6809 & 0.0995 & 0.9744 \\ 0.3870 & 0.0000 & 0.8236 & 0.0646 \\ 0.6202 & 0.2840 & 0.7098 & 0.4588 \\ 0.2801 & 0.1482 & 0.2776 & 0.2322 \end{pmatrix}$$

En el ejemplo vemos que hemos usado una aproximación de rango 2 a la matriz original y que el resultado es una aproximación bastante buena. Sabemos que no es la mejor aproximación ya que ésta es la que devuelve la SVD pero la diferencia es que estas matrices solo pueden tener números positivos.

El estudio de la convergencia de este algoritmo es complicado. Al estar basado en multiplicaciones está garantizado que las matrices siempre serán no-negativas. Por el mismo motivo una vez que un elemento de alguna de las dos matrices se hace igual a cero queda en cero en la matriz resultado. Esto es bueno porque contribuye a que el resultado tienda a generar matrices relativamente dispersas pero por otro lado es malo ya que una vez que el algoritmo inicia su proceso ésta “condenado” a llegar a una única solución posible.

Esta solución es totalmente dependiente de la inicialización por lo que diferentes ejecuciones del algoritmo pueden generar resultados bastante distintos, es decir que tenemos una importante tasa de incertidumbre. Es bueno, en caso de ser posible, agregar restricciones a NMF y luego proceder por Gradient Descent.

14.8.2 La Base de NMF

Al igual que la SVD o K-Means NMF descompone una matriz en una base y coordenadas en esa base, la base tiene k vectores de igual que los datos. En el caso de NMF la base y las coordenadas solo pueden tener elementos positivos. Esto obliga al algoritmo a descomponer cada vector del set de datos original en una suma de partes. Cada vector en la base debe ser entonces una “parte” del set de datos original de forma tal que sumando partes de los diferentes vectores logremos aproximar la reconstrucción del set de datos original.

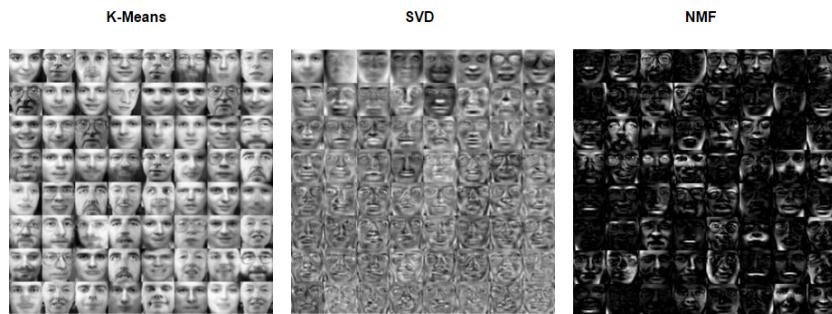


Figure 14.46: Comparacion de SVD, K-Means y NMF

En esta comparación podemos ver los vectores en la base de K-Means, SVD y NMF para un set de imágenes con rostros. Como vemos los vectores de la base en K-Means parecen rostros del set de datos original, esto es porque cada rostro es un promedio de otros rostros, en general similares. En SVD los rostros son fantasmagóricos, algunos están en negativo porque las coordenadas de la descomposición de la SVD pueden ser negativas, los vectores además están

ordenados por importancia, el primero es el mas importante para reconstruir cualquier rostro del set de datos, el segundo es el segundo mas importante, etc. Finalmente NFM nos muestra imágenes muy oscuras, en donde cada imagen tiene una zona mas clara, estas son las diferentes "partes" que luego podemos sumar para reconstruir cualquier rostro de nuestro set de datos original.

En cuanto a poder de reconstrucción SVD suele dar los mejores resultados, esto es lógico pues aprendimos que es la mejor aproximación de rango "k" a la matriz original. La desventaja de SVD es que la base que genera no es muy útil y la reconstrucción es difícil de interpretar al ser una combinación lineal de la base.

K-Means es el método con peor nivel de reconstrucción pero con la base que más se asemeja a las imágenes originales. Esto es lógico ya que cada imagen debe quedar asociada al centroide más similar por lo que los centroides son un promedio de las imágenes dentro del cluster.

Finalmente NMF es una tercer variante en la cual la base representa las partes de un determinado objeto y la reconstrucción se hace tomando diferentes partes de la base. Esto es así gracias a la restricción de no-negatividad de las matrices generadas por NMF.

	Kmeans	SVD	NMF
Base	Promedio de puntos	Ortonormal	Partes
Representacion	1 elemento de la base	CL	Sumatoria
Poder Reconstructivo	Muy bajo	Muy alto	Intermedio
Semejanza base-datos	Muy alta	Muy baja	Partes
Vectores ordenados por importancia	No	Si	No

Table 14.1: SVD, Kmeans y NMF

Veamos la base de NMF para MNIST:

Estas son las partes de los trazos a partir de las cuales podemos reconstruir cualquier dígito. Podemos analizar el poder de reconstrucción de NMF.

Se puede ver claramente como las bases de mayor cantidad de vectores logran una mejor reconstrucción de los datos originales. Esto no es ni más ni menos que una forma de compresión destructiva de los datos, mediante las matrices W y H representamos a los datos originales en mucho menos espacio. Por ejemplo una matriz de 40×2 y una matriz de 2×40 suman en total 160 elementos que son muchos menos que los 40×40 necesarios para almacenar los datos originales. Esto funciona para cualquier tipo de dato!. Es decir que tanto K-Means como SVD como NMF pueden funcionar como algoritmos de compresión de datos o para reducir dimensiones.

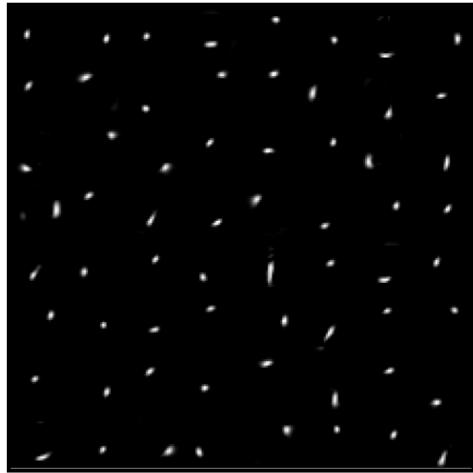


Figure 14.47: Base de NMF para MNIST

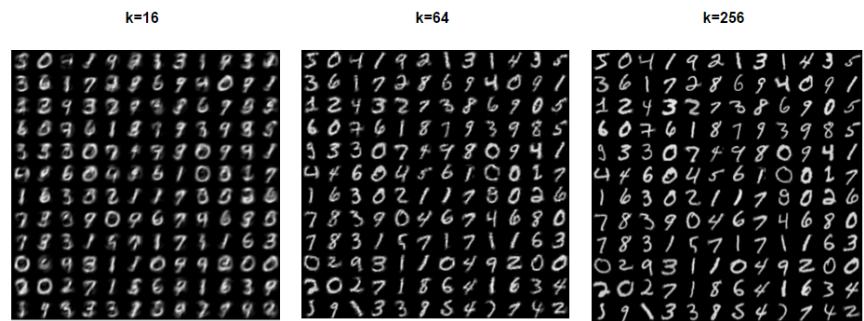


Figure 14.48: Reconstrucción de MNIST con NMF

Chapter 15

PageRank y sus Derivados

It's quite complicated and sounds circular, but we've worked out a way of calculate a Web site's importance

- Larry Page

PageRank [Page-Brin] es uno de los algoritmos mas importantes en la historia de la computación, es el algoritmo que dio origen a Google y permitió que los Search Engines le ganen la pulseada a los spammers en una batalla que continua hasta el día de hoy. PageRank está basado en principios matemáticos muy simples y elegantes. Es un algoritmo que tiene muchísimas variantes y algoritmos derivados, en este capítulo vamos a estudiar el fascinante mundo de PageRank y muchos de sus derivados .

En un principio los motores dedicados a buscar páginas web trabajaban analizando pura y exclusivamente el contenido de estas páginas. De acuerdo a la consulta del usuario se asignaba un puntaje a cada página que dependía de cosas como la cantidad de veces que los términos buscados aparecían en la página, la distancia entre los mismos, el lugar en que aparecían los términos, etc.

Lamentablemente este método es sensible a que los usuarios modifiquen, maliciosamente, el contenido de sus páginas para quedar mejor posicionados en los resultados de las búsquedas. Por ejemplo para una cierta búsqueda podemos ver que página queda primera, copiar su contenido y pegarlo a nuestra página en texto oculto, luego de esto deberíamos ranquear en los primeros lugares. Otra técnica común era repetir en texto oculto una cierta palabra miles o millones de veces, por ejemplo ponemos "movies" en texto invisible 1 millón de veces en nuestra página y cuando el usuario busque "movies" nuestra página estará entre las primeras aunque ni siquiera se trate sobre ese tema.

El gran éxito de Google fue lograr ranquear los resultados de las búsquedas de forma independiente del contenido de las mismas y eso fue logrado con el

algoritmo PageRank.

PageRank se basa en la estructura de links de la Web. El concepto básico es que cada página tiene una cierta "importancia" que es intrínseca y depende de los links que lleven a dicha página. Cuántos mas links nos puedan llevar a una cierta página mas importante será la misma.

Para entender la idea detrás de PageRank vamos a comenzar con el modelo matemático detrás de PageRank que es el modelo de Random surfers.

15.1 Random Surfers

Un *Random Surfer* es un navegante aleatorio en la web. Este navegante comienza en cualquier página al azar, desde esa página nuestro navegante elige un link al azar y navega a la página, este proceso se repite indefinidamente. En su camino aleatorio por la web el navegante va pasando por las distintas páginas, el concepto detrás de pagerank es darle a cada página un peso que es igual a la probabilidad de que nuestro random surfer termine su recorrido en esa página luego de n saltos.

El modelo de random surfers no es ni mas ni menos que un modelo de Markov en donde tenemos tantos nodos como páginas y aristas como links. La probabilidad de cada arista es 1 sobre el total de links en la página. Es decir que si una página tiene dos links cada uno tiene una probabilidad de 1/2.

Ejemplo: Supongamos que tenemos tres páginas: A,B y C. La página A tiene links a B y C. la página B solo linkea a A y la página C solo linkea a B. Nuestra cadena de Markov tiene el siguiente aspecto.

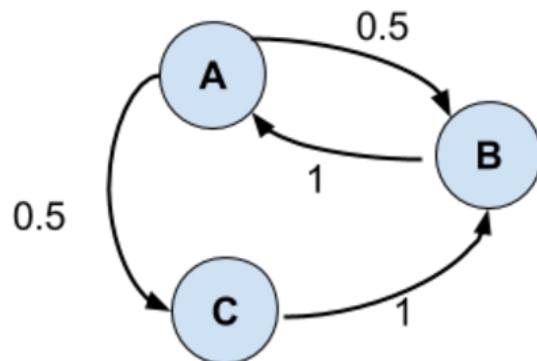


Figure 15.1: Ejemplo Markov Chain

Inicialmente la probabilidad de cada página es 1/3 ya que nuestro navegante aleatorio puede empezar en cualquier página, esto lo representamos mediante el vector $[1/3, 1/3, 1/3]$

La matriz de la cadena de Markov es la siguiente:

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1/2 & 0 & 1 \\ 1/2 & 0 & 0 \end{pmatrix} \quad (15.1)$$

Notar que hemos puesto los links que salen de cada página por columnas. Desde A podemos ir a B y C, desde B solo a A y desde C solo a B.

Nuestro objetivo es ahora calcular la probabilidad de cada página luego del primer salto.

Si comenzamos en A (prob 1/3) podemos ir a B o C entonces la probabilidad de A va a ser cero (no podemos quedarnos), la de B 1/6 y la de C 1/6. Por otro lado comenzando en B (prob 1/3) solo podemos ir a A entonces la prob de A va a ser 1/3 la de B 0 y la de C 0 Finalmente empezando desde C las probabilidades son 0,1/3,0.

La probabilidad de caer en A luego de un link es igual a la probabilidad de empezar en A y quedar en A (0) mas la probabilidad de empezar en B e ir a A (1/3) mas la probabilidad de empezar en C y pasar a A (0). Por lo tanto la probabilidad de A luego de 1 salto es 1/3. Observemos que esto es igual al producto de la fila 1 de la matriz por la probabilidad de A, ya que la fila 1 de la matriz tiene las probabilidades con las cuales se puede llegar desde otras páginas hasta A.

De la misma forma la probabilidad de B luego de un salto surge de multiplicar la fila 2 de la matriz por la probabilidad de B y nos da 1/2.

Finalmente la probabilidad de C es 1/6.

Por lo tanto luego de un salto las probabilidades de cada página se pueden calcular como el producto de la matriz de Markov por las probabilidades iniciales.

$$\begin{pmatrix} 0 & 1 & 0 \\ 1/2 & 0 & 1 \\ 1/2 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} = \begin{pmatrix} 1/3 \\ 1/2 \\ 1/6 \end{pmatrix} \quad (15.2)$$

Observemos que la suma de probabilidades es 1 lo cual es lógico ya que todas las columnas de la matriz original suman 1.

Si queremos averiguar las probabilidades luego de 2 saltos tomamos el resultado anterior [1/3,1/2,1/6] y lo usamos como las probabilidades iniciales, es decir que volvemos a multiplicar:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1/2 & 0 & 1 \\ 1/2 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1/3 \\ 1/2 \\ 1/6 \end{pmatrix} = \begin{pmatrix} 1/2 \\ 1/3 \\ 1/6 \end{pmatrix} \quad (15.3)$$

Según parece luego de dos saltos la probabilidad de la página A es del 50%. El PageRank se define como las probabilidades luego de infinitos saltos. Esto

nos lleva a pensar si PageRank converge y si converge a que valor lo hace, por lo tanto vamos a tener que desarrollar algo de teoría.

15.2 Convergencia

Theorem 29 (Distribución Estacionaria) *Si A es una matriz estocástica entonces existe v tal que:*

$$Av = v$$

Una matriz estocástica es aquella en la cual todas sus filas o columnas suman 1, en el caso de PageRank por definición todas las columnas suman 1 y por lo tanto la matriz es, en principio, estocástica. Existen algunos casos patológicos en los cuales la matriz no es estocástica de los que vamos a hablar mas adelante.

Queremos probar que existe v tal que

$$Av = v$$

lo cual demostraría que existe una distribución de probabilidades estacionaria, es decir que no cambia al pre-multiplicarla por la matriz A .

Si $Av = v$ entonces la matriz A tiene un autovalor igual a 1. Queremos mostrar entonces que una matriz estocástica tiene un autovalor igual a 1.

Theorem 30 (Perron-Frobenius) *El autovalor principal de una matriz estocástica es igual a 1*

Para demostrar esto primero vamos a demostrar que 1 es un autovalor y luego demostraremos que no puede existir un autovalor mayor.

Sabemos que todas las columnas suman 1 por lo tanto en $A - I$ todas las columnas suman cero. Esto muestra que $A - I$ es singular. Por lo tanto existe una combinación lineal no trivial de sus columnas v que nos da el vector nulo. Por lo tanto 1 es un autovalor y v es el autovector asociado.

Hemos demostrado que A tiene un autovalor igual a 1. Veamos ahora que no puede haber un autovalor mayor a 1.

Supongamos que existe $\lambda > 1$ tal que $Ax = \lambda x$. Como todas las columnas de A suman 1 ningún elemento en Ax puede ser mayor al elemento máximo de x , por otro lado si $\lambda > 1$ entonces algún elemento de λx es mayor al elemento máximo de x , por lo tanto llegamos a un absurdo. Es simple demostrar que el autovalor 1 es único, si existiera otro autovalor igual a 1 entonces tendríamos que para $v_1 \neq v_2$ $Av_1 = v_1$ y $Av_2 = v_2$ y entonces $v_1 v_2$ tienen que ser iguales.

Hemos demostrado entonces que una matriz estocástica tiene un único autovalor igual a 1 y que es el autovalor principal de la matriz.

Ahora vamos a demostrar que el método iterativo que hemos propuesto converge a la distribución estacionaria.

Theorem 31 (Power Method) *Sea A una matriz diagonalizable y sea x_0 un vector aleatorio si repetimos $x_0 = Ax_0$ en el límite x_0 es el autovector asociado al autovalor de mayor valor absoluto de la matriz*

Notemos que este teorema sirve para cualquier matriz, no solo para matrices estocásticas

Para demostrar esto vamos a escribir a x_0 como una combinación lineal de los autovectores de A :

$$x_0 = \alpha_1 v_1 + \dots + \alpha_n v_n$$

Si calculamos $x_t = A * x_0$ tenemos:

$$x_t = \alpha_1 \lambda_1^{(t)} v_1 + \dots + \alpha_n \lambda_n^{(t)} v_n$$

Si suponemos que λ_1 es mayor a λ_2 entonces el coeficiente de v_1 crece más rápido que los demás coeficientes y luego de varias iteraciones domina la fórmula, por lo tanto luego de iterar:

$$x_t = \alpha_1 \lambda_1^{(t)} v_1$$

Como v_1 es un autovector entonces $\alpha_1 \lambda_1^{(t)} v_1$ también lo es y como λ_1 tiene que ser el autovalor de mayor valor absoluto para dominar la fórmula entonces v_1 es un autovector asociado a dicho autovalor.

Por lo tanto mediante el método iterativo encontramos el autovalor principal de una matriz y un autovector asociado al mismo.

En el caso de una matriz estocástica hemos probado que el autovalor principal es 1 por lo tanto el autovector asociado es aquel que cumple que $Av = 1v$ por lo tanto $Av = v$ y esto prueba que el método converge.

En conclusión la matriz de PageRank tiene un único autovalor principal igual a 1 y el método iterativo converge a un autovector asociado a dicho autovalor.

En nuestro ejemplo si repetimos el método iterativo llegamos a que $PR(A)=0.4$, $PR(B)=0.4$, $PR(C)=0.2$. Si observamos nuestro modelo de Markov podemos analizar que esto tiene sentido ya que podemos ir de A hacia B o de B hacia A pero a C solo podemos llegar desde A, es decir que es lógico que la probabilidad de A sea igual a la de B y que estas probabilidades sean mayores a las de C.

Finalmente en base a todos los teoremas anteriores podemos enunciar el último eslabón que nos faltaba.

Theorem 32 (Teorema Fundamental de las Cadenas de Markov) *Para un Random-Walk suficientemente largo en una cadena de Markov la probabilidad de terminar en un cierto vértice "v" es independiente del vértice en el cual se inició el Random Walk.*

Este teorema se desprende del teorema de la distribución estacionaria que ya hemos visto y es muy importante porque es la pieza final en el rompecabezas de PageRank, mediante un Random-Walk lo suficientemente largo podemos calcular la probabilidad de terminar en cada una de las páginas de nuestro grafo y al ser esto independiente de donde comenzamos nos sirve como una medida de importancia de cada página.

Hemos pasado por varios teoremas pero lo bueno es que ya estamos en condiciones de entender todo el capítulo, PageRank en cualquiera de sus variantes

es un algoritmo que busca el autovector principal de una matriz estocástica, si entendemos esto ya entendimos todo el algoritmo.

15.3 Dead Ends

Consideremos el siguiente grafo:

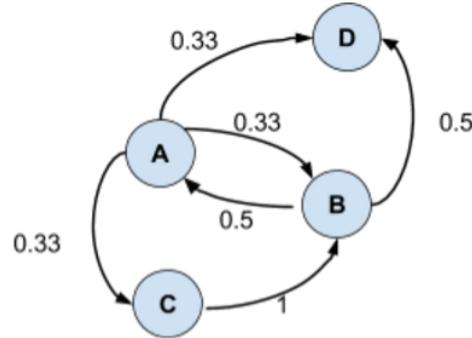


Figure 15.2: Dead Ends

Y su matriz asociada:

$$\begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0.33 & 0 & 1 & 0 \\ 0.33 & 0 & 0 & 0 \\ 0.33 & 0.5 & 0 & 0 \end{pmatrix} \quad (15.4)$$

La página D es un *dead-end* es decir una página desde la cual no se puede ir a ninguna otra página. Nuestro navegante aleatorio eventualmente va a caer en D y quedará atrapado. Por lo tanto D va a capturar todo el PageRank y las otras páginas van a quedar en cero. El problema es que la matriz no es estocástica, para que PR funcione la matriz **siempre** tiene que ser estocástica. Cuando una página es un dead-end tenemos que corregirla y la forma de hacerlo es agregando links a todas las otras páginas con probabilidad $1/n$.

$$A = \begin{pmatrix} 0 & 0.5 & 0 & 1/4 \\ 0.33 & 0 & 1 & 1/4 \\ 0.33 & 0 & 0 & 1/4 \\ 0.33 & 0.5 & 0 & 1/4 \end{pmatrix} \quad (15.5)$$

Luego de la corrección el PR final de cada página converge a: [0.2308, 0.3077, 0.1538, 0.3077], es decir que las páginas mas importantes son B y D, luego A y finalmente C.

Siempre que encontremos dead-ends el primer paso es corregirlos para asegurar que la matriz sea estocástica.

15.4 Spider Traps y Teletransportación

Consideremos ahora el siguiente grafo:

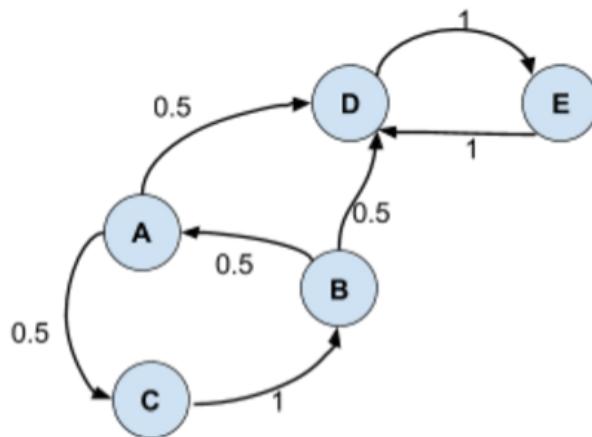


Figure 15.3: Dead Ends

Y su matriz asociada:

$$A = \begin{pmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad (15.6)$$

Podemos ver en el en grafo que una vez que nuestro navegante aleatorio llegue a D va a quedar atrapado en D y E sin poder salir, por lo tanto el PageRank va a distribuirse entre D y E y el resto de las páginas va a quedar en cero. Esto es lo que llamamos un *Spider Trap*. El término hace alusión a los robots indexadores de los buscadores que se llaman "spiders", una vez que un robot encuentre el

link A-D o B-D y lo navegue no va a poder salir del ciclo D-E-D-E... es decir que queda encerrado en una trampa.

Si observamos la matriz asociada al grafo veremos que es estocástica por lo que en principio no hace falta hacerle ninguna corrección. La solución a este problema fue el gran descubrimiento del algoritmo PageRank y se llama *teletransportación*.

Cada vez que nuestro navegante aleatorio llegue a una página con probabilidad β lo vamos a hacer elegir un link al azar y seguirlo y con probabilidad $1 - \beta$ lo vamos a teletransportar a cualquier página al azar (todas equiprobables).

En nuestro grafo el navegante aleatorio va a llegar a D y puede que haga algunas veces el ciclo D-E-D-E... pero eventualmente se va a teletransportar a otra página y va a poder salir del loop.

Tenemos que ver como aplicar la teletransportación a la matriz de PR, y es algo muy simple:

$$A = \beta \begin{pmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} + (1 - \beta) \begin{pmatrix} 1/5 & 1/5 & 1/5 & 1/5 & 1/5 \\ 1/5 & 1/5 & 1/5 & 1/5 & 1/5 \\ 1/5 & 1/5 & 1/5 & 1/5 & 1/5 \\ 1/5 & 1/5 & 1/5 & 1/5 & 1/5 \\ 1/5 & 1/5 & 1/5 & 1/5 & 1/5 \end{pmatrix}$$

Si usamos PR ahora con nuestra nueva matriz A usando $\beta = 0.8$ obtenemos [0.07,0.09,0.07,0.39,0.35], las páginas D y E tienen el mayor PR lo cual es lógico pero ahora el PR de las otras páginas no es cero.

Observemos que el proceso de multiplicar la matriz A por el vector de PR es equivalente a a hacer:

$$\begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix} = \beta \begin{pmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix} + (1 - \beta) \begin{pmatrix} 1/5 \\ 1/5 \\ 1/5 \\ 1/5 \\ 1/5 \end{pmatrix}$$

Donde $[A,B,C,D,E]$ son los PR en alguna de las iteraciones. Es decir que podemos multiplicar el vector de PR por la matriz de links multiplicada por β y luego a cada elemento del vector sumarle $1 - \beta \frac{1}{N}$. Los resultados son exactamente los mismos.

La teletransportación soluciona otro problema que es el caso de los grafos periódicos.

Si el grafo es periódico el PR no converge sino que oscila. Para que esto no ocurra debe ser posible garantizar que luego de un Random-Walk lo suficientemente largo existe una probabilidad positiva de estar en cualquier vértice del grafo en el próximo paso. Una forma de asegurar esto que se usa en Cadenas de Markov es agregando un link en cada nodo a si mismo, pero en PageRank no es necesario hacer esto porque la teletransportación asegura la aperiodicidad del grafo, ya que en cualquier momento con una probabilidad positiva podemos teletransportarnos a cualquier otro nodo.

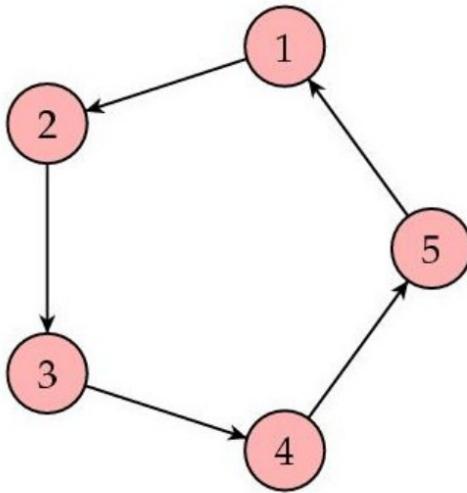


Figure 15.4: Grafo Periódico

Con el agregado de la teletransportación y recordando que es necesario arreglar los dead-ends tenemos desarrollado el algoritmo de PR de forma completa. Vamos a comentar ahora como hacer para que PR funcione cuando el grafo tiene billones de nodos es decir en el caso de Google y la Web.

Interpretando β

El parámetro β es la probabilidad de seguir un link es decir de no teletransportarse. En general se usar $\beta = 0.8$ o similar. Analicemos el efecto de β en nuestro navegante aleatorio. Cuando $\beta = 1$ tenemos el caso sin teletransportación, en este caso hay un único random-walk infinito sobre nuestra cadena de Markov. Si $\beta < 1$ entonces la cadena se reinicia cada tanto, cuanto menor sea β mas random-walks vamos a tener y mas cortos van a ser estos. En el límite cuando $\beta \rightarrow 0$ tenemos infinitos random-walks de longitud 0 es decir que simplemente tomamos nodos al azar de nuestro grafo para visitarlos y el resultado será que el PR de cada uno va a ser $1/N$.

15.5 Page Rank en la práctica

Si tenemos una matriz con billones de nodos no podemos pensar en calcular el producto de la matriz por el vector de PR de las páginas ya que incluso para una sola iteración este proceso sería inviable. Afortunadamente, como en tantos otros casos, hay otra forma de hacer lo mismo. Existe una relación casi simbiótica entre PageRank y MapReduce, esto explica el gran éxito de Google

ya que MapReduce surge, precisamente, como la solución al problema de aplicar PR para toda la Web.

Consideremos de que forma queremos procesar el PR, podemos ir página por página y en cada página tenemos una serie de links a otras páginas, sabemos ademas el PR de cada página. Entonces estando en una página sabemos que esta transfiere su PR a cada una de las páginas linkeadas de forma equiprobable. Podemos entonces simplemente sumar a cada una de las otras páginas el PR correspondiente a la página origen dividido por la cantidad de links y multiplicado por β . Una vez que hicimos esto para todas las páginas simplemente sumamos al vector resultado la teletransportación que es $(1 - \beta) * (1/N)$. Esto nos da un algoritmo muy simple para procesar el PR si tenemos el vector de PR en memoria:

Algorithm 36: PageRank v1.0

```

1  $PR[1..N] = 1/N$  while not convergencia do
2   for  $i=1..N$  do
3     for  $l$  in  $links(i)$  do
4       //transfer PR to linked pages
5        $PR[l] = PR[l] + \beta PR[i]/|l|$ 
6   // Add teleportation to ALL pages
7    $PR = PR + (1 - \beta)(1/N)$ 

```

Observemos que la matriz de links en realidad no hace falta, solo necesitamos el vector con el PR de cada página y por cada página los links que tiene la misma.

Este método es suficiente para grafos realmente grandes pero no para billones de páginas como en el caso de Google, en estos casos es donde podemos usar Map Reduce de una forma muy simple.

15.5.1 PageRank via Map Reduce

En base al procedimiento anterior podemos pensar una versión muy simple de PageRank usando MapReduce, el proceso Map procesa una página y sus links y emite un registro por cada link en la página transfiriéndole la parte del PageRank que le corresponde, el proceso Reduce suma todos los puntajes obtenidos y agrega la teletransportación.

Algorithm 37: PageRank MAP

Data: url,pagerank,listalinks

```

1 for  $l$  in  $listalinks$  do
2   emit(link, $\beta * pagerank/|listalinks|$ )

```

Este proceso se repite tantas veces como sea necesario para llegar a la convergencia, es decir que vamos alternando fases de Map y Reduce en donde cada

Algorithm 38: PageRank Reduce

Data: url,[puntajes]
1 emit(url,sum(puntajes)+(1 - β)/N)

reduce actualiza el valor del PageRank para cada página y sirve como dato para la siguiente fase de Map.

Notemos que si agregamos nuevas páginas podemos incorporarlas simplemente con PR=0 y tarde o temprano el algoritmo va a converger, no es necesario volver a empezar el algoritmo ya que comenzando con cualquier vector el método iterativo siempre converge al autovector asociado al autovalor principal de la matriz.

15.6 TopicRank

El concepto de teletransportación en PageRank es muy importante y permite varias variantes interesantes. En el algoritmo tradicional nos teletransportamos con igual probabilidad a cualquier página, pero esto lo podemos cambiar. Si decidimos, por ejemplo, teletransportarnos solo a aquellas páginas que tratan sobre un determinado tema tenemos una variante de PageRank llamada TopicRank en donde las búsquedas sobre un cierto tema van a darnos como resultado páginas que tratan sobre dicho tema. Esto se logra manipulando el concepto de teletransportación.

Supongamos que sabemos, de alguna forma, cuáles son las páginas que tratan sobre un determinado tema, necesitamos entonces correr una versión de PR para dicho tema, tendremos un PR por cada tema. Para realizar el PR temático en lugar de sumar $(1 - \beta)/N$ a cada página sumamos $(1 - \beta)/|C|$ en donde $|C|$ es la cardinalidad del tema, es decir la cantidad de páginas que están clasificadas dentro del tema que nos interesa.

Tomemos el siguiente ejemplo, si aplicamos PR tradicional con $\beta = 0.8$ llegaremos al siguiente resultado: A=0.2028, B=0.3821, C=0.2840, D=0.1311. Supongamos ahora que las páginas temáticas son A y D entonces tenemos:

El cálculo del PR en este caso se realiza mediante la siguiente fórmula:

$$r = \beta \begin{pmatrix} 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1/2 & 1/2 & 0 & 0 \\ 1/2 & 0 & 0 & 0 \end{pmatrix} r + (1 - \beta) \begin{pmatrix} 1/2 \\ 0 \\ 0 \\ 1/2 \end{pmatrix}$$

Iterando llegamos a A=0.2358, B=0.3396, C=0.2302, D=0.1943. Podemos ver que el PR de A y D es mayor al calculado antes. Si queremos darle mayor peso todavía a las páginas temáticas tenemos que darle mayor probabilidad a la teletransportación es decir usar un β mas chico. Usando $\beta = 0.5$ obtenemos A=0.3077, B=0.2308, C=1346, D=3269 en donde las páginas temáticas son las que tienen mayor PR.

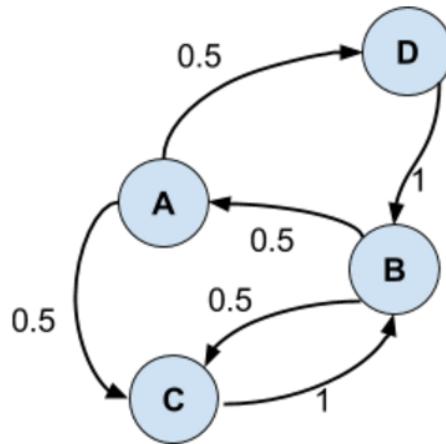


Figure 15.5: Topic Rank 1

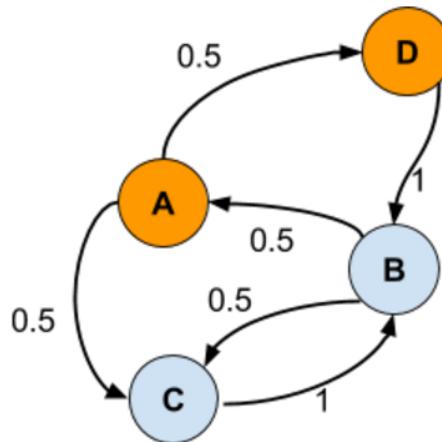


Figure 15.6: Topic Rank 2

15.7 TrustRank

TrustRank es una variante de PageRank para lograr disminuir el efecto de las *spamfarms*. El objetivo de una SpamFarm es aumentar, el PR de una determinada página. Para lograr esto los spammers crean una gran cantidad de páginas que linkean a la página en cuestión y links desde la página a estas otras, for-

mado una "red". Luego insertan links en páginas públicas que lleven a la página objetivo.

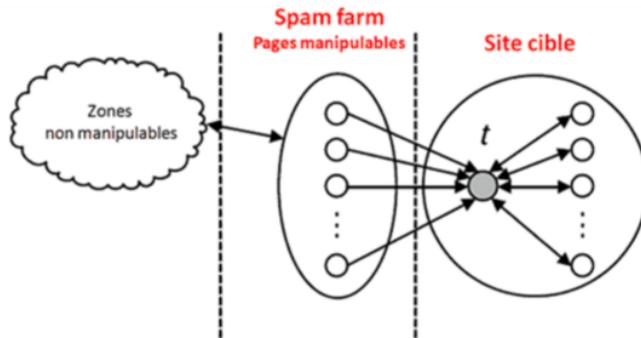


Figure 15.7: Spam Farms

Los links a la página se pueden lograr insertando links en comentarios en sitios confiables que admiten comentarios de los usuarios, por ejemplo los comentarios de noticias en algún diario, foros, comentarios en Redes Sociales, etc. Todos estos links públicos permiten que los robots que indexan las páginas descubran la página objetivo. Una vez que esto ocurre el robot va a descubrir los links desde esta página a todas las que están dentro del spam farm y el link desde estas a la página objetivo. Con algo de trabajo este método logra aumentar artificialmente el PR de una página.

La forma en que TrustRank combate los SpamFarms es mediante la manipulación de la teletransportación. La idea es simple: solo nos podemos teletransportar a páginas confiables. Esto es equivalente a tener un TopicRank en donde el "topic" es que la página es confiable.

Determinar qué páginas son confiables no es una tarea sencilla, podemos definir ciertos dominios como confiables como por ejemplo .gov, .edu, .mil, etc. Y luego para el resto de los dominios recurrir a la clasificación manual.

Al usar TrustRank cada página tiene un valor de TrustRank y un valor de PageRank, lo que nos importa es la diferencia entre ambos. Si llamamos TR al valor del TrustRank y PR al valor del pagerank podemos determinar la "masa de spam" de una página como:

$$\text{SpamMass} = \frac{PR - TR}{PR} \quad (15.7)$$

A partir de la masa de spam podemos establecer un cierto umbral y eliminar las páginas que estén por encima del mismo. O flaggear las páginas para que un humano las supervise y decida si deben ser eliminadas del índice o no.

15.8 SimRank

Continuando con las variantes posibles para la teletransportación tenemos el caso en el cual solo podemos teletransportarnos a una única página. Lo que obtenemos entonces es una serie de random-walks en los cuales siempre volvemos al mismo punto de partida. El PageRank de cada página representa entonces la probabilidad de llegar a cada página desde una cierta página origen y a esto lo llamamos *SimRank*

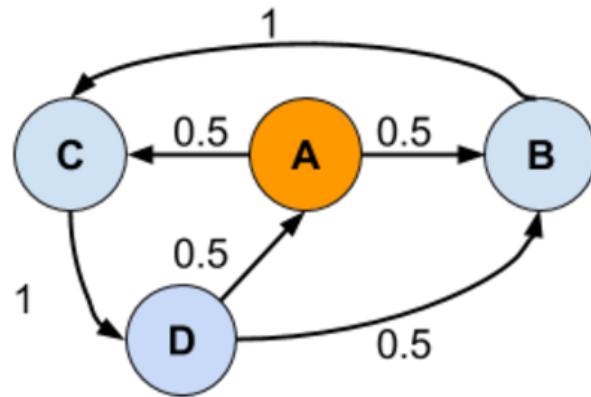


Figure 15.8: Sim Rank

En el ejemplo tenemos un grafo en donde queremos calcular el SimRank con respecto a la página A, el proceso será igual al que hicimos hasta ahora, en la teletransportación vamos a sumarle $(1 - \beta)$ a la página A únicamente. Es decir que cualquier teletransportación nos lleva a "A".

$$r = \beta \begin{pmatrix} 0 & 0 & 0 & 1/2 \\ 1/2 & 0 & 0 & 1/2 \\ 1/2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} r + (1 - \beta) \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

El resultado es A=0.2897, B=0.2056, C=0.2804, D=0.2243. El PR de A no nos interesa, lo que nos interesa es que la página mas "similar" a A es la C, luego D y finalmente B. Es interesante tomarnos unos minutos e interpretar este resultado mirando el grafo.

15.8.1 Sim Rank via Montecarlo

EL problema del algoritmo SimRank es que no podemos calcular el SimRank de cada página, esto tomaría un total de N ejecuciones de SimRank lo cual, en general, no es aceptable. Además no es algo que tenga sentido hacer ya que no siempre necesitamos todas las semejanzas sino solo algunas pocas.

La solución pasa por no calcular el SimRank sino por simularlo mediante el método de Montecarlo. Cuando queremos obtener las páginas mas similares a alguna hacemos una simulación de Random-Walks comenzando en la página y con un cierto β en donde $1-\beta$ es la probabilidad de re-iniciar el RandomWalk. A medida que realizamos los Random-Walks registramos por que páginas pasamos y cuántas veces visitamos cada una. Luego de simular una buena cantidad de RWs, lo cual puede hacerse muy rápidamente, tenemos una muy buena aproximación al SimRank de la página simplemente listando las demás en orden decreciente por cantidad de visitas.

Este método se puede usar en Redes Sociales para recomendar amigos o usuarios a seguir, es el método usado por Twitter para recomendar usuarios a seguir, cuando un usuario visita Twitter es sencillo simular n Random-Walks a partir del usuario y contar las veces que pasamos por otros usuarios, luego simplemente recomendamos los que mas veces visitamos y no son seguidos por el usuario. El método tiene resultado muy buenos y consume muy pocos recursos ya que funciona en memoria sobre el grafo completo de la Red Social.

15.8.2 Panther

Existe un algoritmo muy similar a la versión MonteCarlo de SimRank que tiene propiedades muy diferentes denominado Panther [Panther]. En este algoritmo creamos Random Walks al azar pero a partir de cualquier nodo, no solo a partir del nodo que nos interesa. La semejanza entre dos nodos se define luego simplemente como el porcentaje de caminos en los cuáles los nodos aparecen juntos.

Por ejemplo si tenemos los siguientes caminos:

1-3-5-2-3-4
1-2-6-7-2-3
3-7-5-8-2-3

La semejanza entre los nodos 3 y 5 será $2/3$ ya que aparecen juntos en dos de los tres paths generados. Este algoritmo sirve para calcular la semejanza entre dos nodos cualesquiera usando Random Walks y una estructura de datos adecuada para contabilizar la cantidad de veces que dos nodos aparecen juntos, por ejemplo puede usarse un índice invertido para indicar en que walks participa cada nodo. Luego dados dos nodos tomamos las listas de walks y contamos la intersección entre las listas como la cantidad de walks en los que ambos nodos aparecen, dividiendo por la cantidad total de walks tenemos la semejanza entre dos nodos cualesquiera.

El método es muy eficiente para redes de incluso billones de nodos.

15.9 VisualRank

VisualRank es un algoritmo para rankear imágenes en un buscador de imágenes. Supongamos que un usuario busca "Mona Lisa" en un buscador de imágenes,

podemos buscar imágenes en las cuales aparezca la frase "Mona Lisa", ya sea en su título, o en algún metadato. El resultado es una gran cantidad de imágenes que parecen responder a la consulta "Mona Lisa" pero no tenemos idea de cuál es realmente la mas relevante. En imágenes no tenemos TF-IDF ni nada por el estilo, entre las imágenes que tienen en su título la frase buscada podemos tener las mas relevantes y las menos relevantes al mismo tiempo.

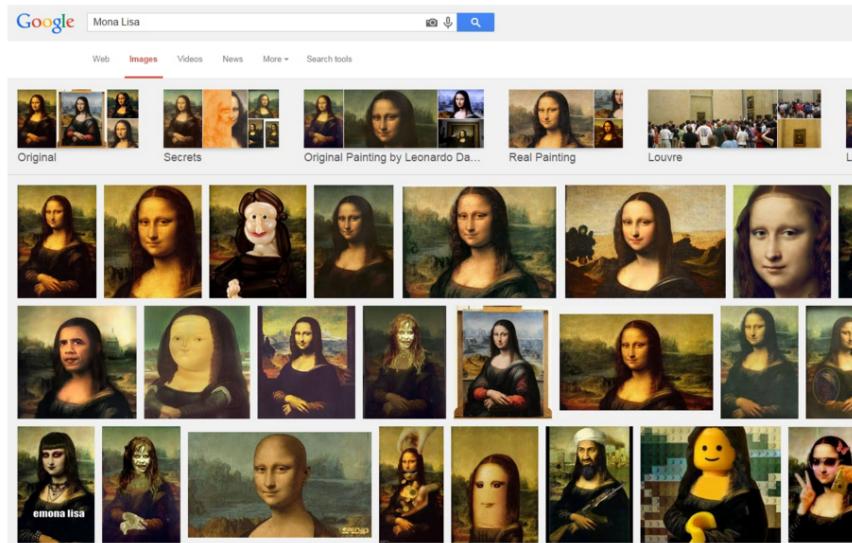


Figure 15.9: Visual Rank

El algoritmo VisualRank se encarga de resolver este problema mediante una ingeniosa idea: la imagen mas relevante debe ser aquella que mas se parece a todas las demás, si podemos rankear las imágenes de acuerdo a su semejanza contra todo el resto tendremos un orden muy efectivo para presentarle los resultados al usuario.

VisualRank comienza armando un grafo entre todas las imágenes recuperadas, un grafo completo en donde cada arista representa la semejanza entre las imágenes. Luego corriendo PageRank en el grafo, usando la semejanza entre imágenes como la probabilidad de visitar una luego de otra, podemos rankear las imágenes y presentar el resultado final. El problema es como calcular la semejanza entre las imágenes.

Para calcular la semejanza entre dos imágenes VisualRank hace uso del algoritmo SIFT (Scale Invariant Feature Transform). Este algoritmo extrae de cada imagen un conjunto de puntos que no se modifican con diferentes escalas, rotaciones o iluminación.

Cada punto extraído por SIFT tiene un descriptor de 128 elementos (un vector). La semejanza entre dos imágenes se calcula mediante la distancia entre todos los puntos SIFT de una imagen contra todos los puntos SIFT de la otra.

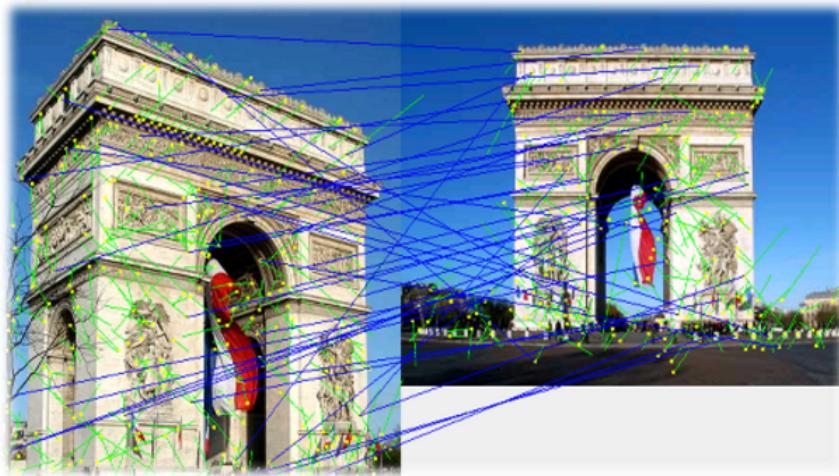


Figure 15.10: SIFT

Ocurre que comparar todos los puntos SIFT contra todos por cada par de imágenes es una tarea muy costosa por lo que VisualRank usa LSH para poder realizar estas comparaciones en $O(1)$, aplicamos una función LSH a los puntos SIFT de dos imágenes y la semejanza es simplemente la cantidad de colisiones que tuvimos en total dividiendo por el total de puntos SIFT.

VusalRank o variantes de VisualRank son usados por los buscadores de imágenes modernos para poder ordenar los resultados de una consulta por relevancia.

15.10 TextRank

Habiendo aplicado PageRank a imágenes podemos, ¿por qué no?, aplicar el algoritmo sobre textos. El resultado de este experimento se llama TextRank y es uno de los algoritmos mas efectivos para realizar resúmenes automáticos. Un resumen automático consiste en extraer de un texto sus frases mas significativas y presentarlas luego en orden de aparición.

Por ejemplo aquí mostramos un resumen automático del guión de Star Wars realizado por TextRank.

The Jedi Council led by Yoda reluctantly allows Obi-Wan to train Anakin after Qui-Gon is killed by Palpatine's first apprentice Darth Maul during the Battle of Naboo... When Luke asks about his father whom he has never met Obi-Wan tells him that

Anakin Skywalker was a great Jedi who was betrayed and murdered by Vader...

Para extraer las frases mas significativas TextRank arma un grafo con todas las frases del texto y usa alguna métrica de distancia entre frases para calcular la distancia entre frases, por ejemplo da buenos resultados usar Tf-IDF o BM25. Con el grafo construido TextRank simplemente corre PageRank sobre el mismo y luego extrae las k frases de mayor PageRank para realizar el resumen.

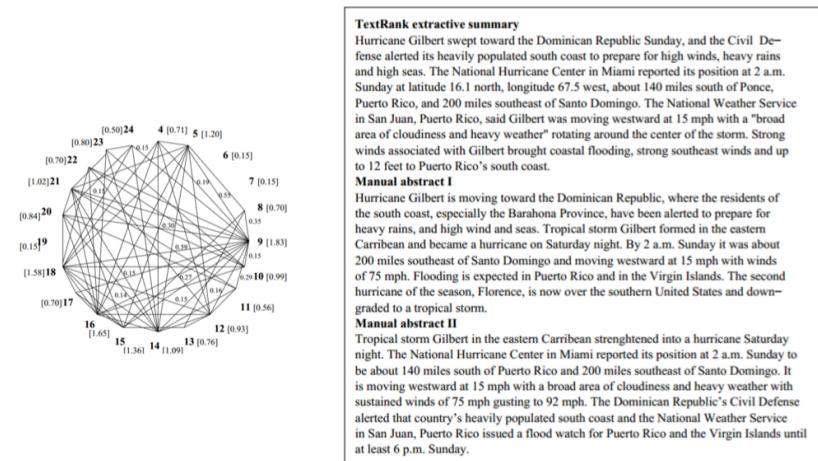


Figure 15.11: TextRank

Este sencillo proceso permite realizar una tarea que para el público en general parece un milagro de la Inteligencia Artificial.

Si en lugar de frases extraemos palabras de un texto, tenemos un algoritmo para extraer los "keywords" mas importantes de nuestro texto. Para la semejanza entre palabras podemos usar la cantidad de veces que las palabras co-ocurren, o la distancia promedio entre las palabras cuando ocurren en la misma frase. TextRank tiene muy buenos resultados como forma de extraer keywords pero es necesario filtrar cuidadosamente los stopwords.

15.11 HITS

HITS [Kleinberg] es un algoritmo para calcular la importancia de los nodos de un grafo, fue publicado casi al mismo tiempo que PageRank y tiene resultados muy similares sin embargo, como es evidente, tuvo un destino muy diferente al de PageRank.

La idea de HITS es calcular dos puntajes para cada página. Un puntaje como "Hub" y un puntaje como "Authority". Un Hub es una página que linkea a buenas páginas, por ejemplo un índice de Wikipedia. Una "Autoridad" es una página que es linkeada por buenos Hubs, en este sentido el puntaje de "Authority" es similar al PageRank.

Inicialmente cada página tiene valor 1 como Hub y valor 1 como Authority. Luego el algoritmo comienza a iterar. El valor de "authority" de una página es la suma del valor "Hub" de todas las páginas que linkean hacia ella. Una vez que calculamos el valor de Authority de cada página calculamos el valor de Hub de cada página como la suma de los valores de Authority de todas las páginas hacia las cuales la página linkea. Los valores de Hub y Authority se normalizan para que todos sumen 1. Este proceso se repite hasta alcanzarla convergencia.

Veamos un ejemplo

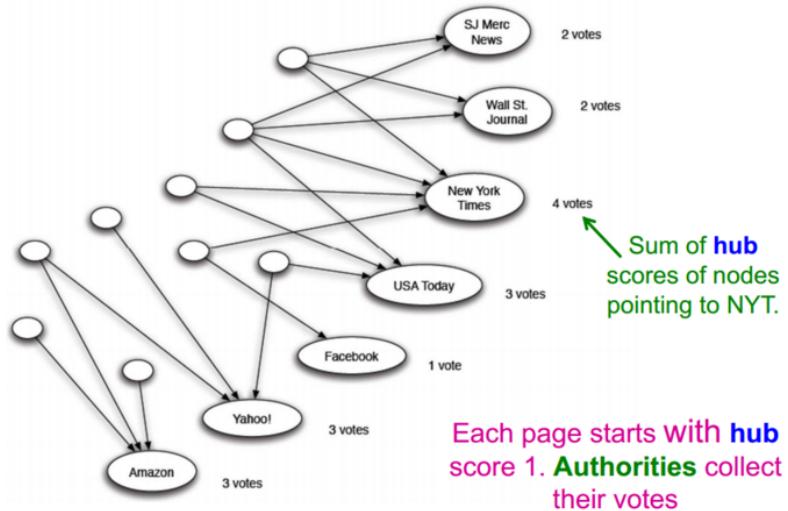


Figure 15.12: HITS paso 1

En el primer paso calculamos el valor de "authority" de cada página que es igual simplemente a la cantidad de páginas que linkean hacia ella ya que inicialmente el valor de "Hub" de cada página es 1. En nuestro ejemplo podemos ver como a "New York Times" le asignamos valor de Authority 4 simplemente porque hay 4 páginas que tienen un link hacia "New York Times". Notemos que el grafo del ejemplo es bipartito solo por razones de simplicidad, el grafo real no tiene porque tener esta restricción.

En el paso dos procedemos en sentido inverso, ahora vamos a calcular el valor como "Hub" de cada página y para eso observamos los links que tiene cada página (salientes) y sumamos los valores de Authority de todas las páginas apuntadas. En el ejemplo el nodo apuntado con valor 11 obtiene sus 11 pun-

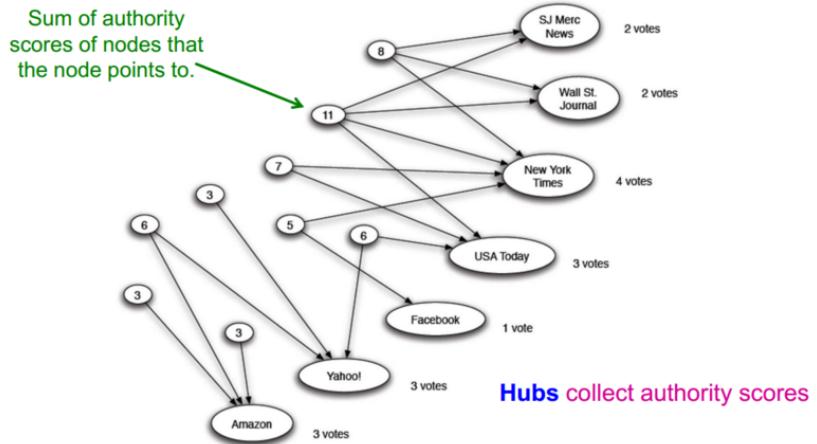


Figure 15.13: HITS paso 2

tos mediante la suma de 2 puntos de "SJ Merc News", 2 puntos de "Wall St Journal", 4 puntos de "NY Times" y 3 puntos de "USA Today"

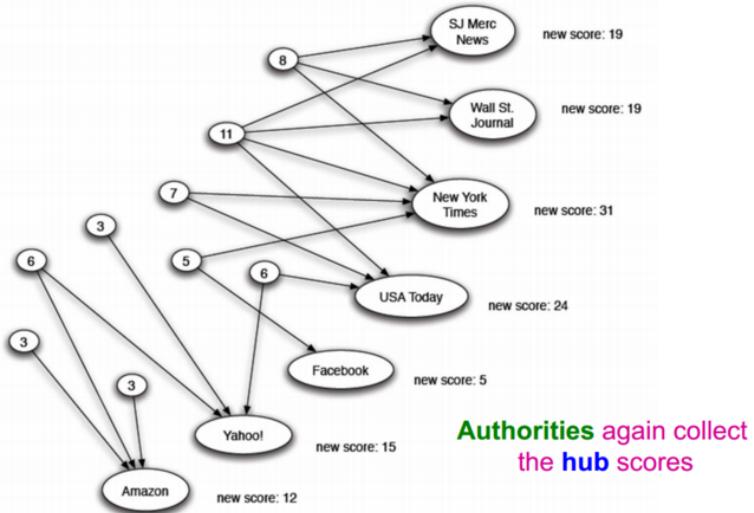


Figure 15.14: HITS paso 3

Luego repetimos el proceso, calculamos el nuevo puntaje como "Authority" de cada página sumando los puntajes como Hubs de los links (entrantes). Vemos

como ahora el New York Times tiene score 31 que surge de los puntajes como Hub de las 4 páginas que lo apuntanL 8,11, 7 y 5.

Theorem 33 (Kleinberg) *HITS normalizando en cada paso converge a un único valor tanto de Hub como de Authority.*

Llamemos A a la matriz de adyacencias del grafo, la matriz de links, en este caso con 0s y 1s indicando si existe o no un link desde una página hacia otra. Esta matriz es diferente a la de PageRank en la cual teníamos probabilidades. Llamaremos h al vector de tamaño N con los valores como Hub de cada página y llamaremos a al vector con los puntajes como Authority de cada página. Tenemos entonces:

$$\begin{aligned} h &= Aa[1] \\ a &= A^t h[2] \end{aligned} \tag{15.8}$$

De [1] en [2] y [2] en [1]:

$$\begin{aligned} h &= AA^t h \\ a &= A^t Aa \end{aligned} \tag{15.9}$$

Es decir que h converge al autovector asociado al autovalor principal de AA^t y a converge al autovector asociado al autovalor principal de $A^t A$ esto demuestra el teorema de la convergencia de HITS. La forma correcta de normalizar HITS para garantizar la convergencia es dividiendo cada puntaje por la raiz cuadrada de la sumatoria del cuadrado de todos los puntajes, es decir obligando a que los vectores h y a tengan norma igual a 1.

En el ejemplo tenemos:

$$\begin{aligned} A &= \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \\ A^t &= \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{aligned} \tag{15.10}$$

El ranking como Hubs nos da Yahoo=0.788, Amazon=0.577, Microsoft=0.211 mientras que el ranking como Authority nos da Yahoo=0.628, Amazon=0.459, Microsoft =0.628. Esto quiere decir que la página que mejor linkea a otras es Yaho, que curiosamente se linkea a si misma(!). Mientras que la mejor página en términos de quien la linkea es un empate entre Yahoo y Microsoft.

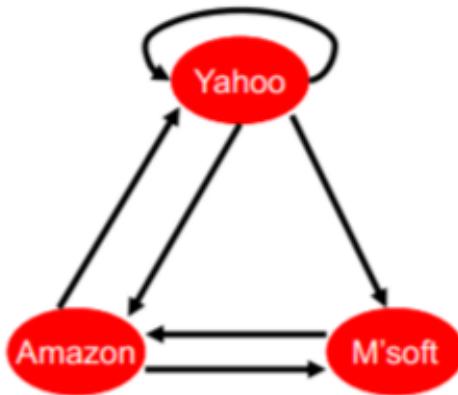


Figure 15.15: HITS Ejemplo

El algoritmo HITS es interesante porque plantea no uno sino dos rankings para cada página, esto nos da la oportunidad de rankear las páginas como fuente de links o de acuerdo a su importancia y también nos permite elaborar esquemas de ranking mixtos en donde usamos una combinación del puntaje como Hub y Authority de la página para el Ranking final.

En cierta forma HITS es mas flexible que PageRank pero sin embargo tuvo muy poco éxito y compararlo contra el suceso de PageRank resultaría redundante. La única aplicación práctica exitosa de HITS fue el buscador "Teoma" que luego fue adquirido por "Ask.com".

Chapter 16

Redes Sociales

By giving people the power to
share we're making the world
more transparent

- Mark Zuckerberg

En este capítulo vamos a estudiar el análisis de Redes Sociales y los algoritmos que podemos usar para obtener información a partir de ellas. Una Red Social, es, básicamente, un grafo en el cual los nodos son usuarios y las aristas expresan las relaciones entre los mismos. En algunas redes las relaciones son dirigidas, por ejemplo en Twitter un usuario puede seguir a otro pero la relación puede no ser recíproca. En Facebook la relación de amistad no tiene dirección, una vez que se acepta la amistad la misma queda expresada mediante una arista no dirigida. En otras situaciones las aristas pueden tener signos o valores, por ejemplo en las Redes en las cuales se puede votar a favor o en contra de un usuario.

En definitiva cada Red Social se presenta como un grafo o red y por eso este capítulo es un estudio sobre grafos y redes, vamos a comenzar entonces con un repaso de propiedades interesantes que podemos obtener a partir de un grafo para luego ver de que forma podemos caracterizar a una Red Social.

16.1 Propiedades de los Grafos

16.1.1 Propiedades Básicas

Vamos a llamar indistintamente nodo o vértice a cada usuario de la Red Social. Vamos a llamar N a la cantidad de nodos del grafo y E a la cantidad de aristas. Llamaremos k_i al grado de cada nodo. Vamos a llamar k al grado promedio del grafo. Si el grafo no es dirigido entonces $k = 2E/N$.

Un grafo *completo* es aquel que tiene todas sus aristas, en un grafo no dirigido esto quiere decir que $k = N - 1$ porque cada nodo está conectado con todos los

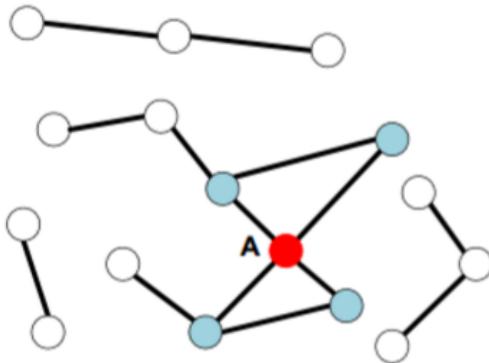


Figure 16.1: Grafos y Redes Sociales

demás y entonces $E = N(N - 1)/2$.

Las Redes Sociales no son grafos completos, por el contrario son grafos dispersos, donde $E \ll E_{max}$ y $k \ll N - 1$. En concreto para la Web $k = 9.64$ lo cual quiere decir que en promedio una página Web tiene 10 links. En LinkedIn $k = 8.87$ es decir que en promedio cada usuario tiene 9 contactos. De la misma forma para la mayoría de las Redes Sociales el grado promedio es un número muy pequeño en relación a la cantidad de nodos en la red.

Esto nos lleva a definir de que forma debemos almacenar el grafo de una Red Social.

16.1.2 Almacenamiento del Grafo

Dado que cada nodo tiene una pequeña cantidad de vecinos la forma más eficiente de almacenar el grafo de una Red Social es mediante una lista de adyacencias en donde por cada nodo almacenamos cuáles son sus nodos vecinos.

Supongamos por ejemplo que nuestra Red Social tiene mil millones de nodos. Necesitamos 30 bits por cada ID de nodo y como el promedio de vecinos es 10 necesitamos unos 300 bits para la lista de adyacencias, esto nos da un total de 330 bits por nodo. En total necesitaríamos unos 41 Gigabytes para almacenar el grafo completo. Notemos que esta cantidad es manejable en memoria por lo que incluso para grafos muy extensos es posible tener el grafo completo de la Red Social en memoria, esto es muy importante ya que en muchos casos no se analiza correctamente el poder de representar los grafos de forma compacta.

16.1.3 Componentes Conexos

En una Red Social en general existe un único componente conexo gigante, esta es una propiedad que tienen todos los grafos con muchos nodos. Es sencillo explicar

el motivo por el cual esto pasa pensando en una Red Social en donde tenemos muchos nodos pero dos componentes conexos. Cuando agregamos una arista, la probabilidad de que la misma conecte dos nodos en distintos componentes conexos es muy alta y tarde o temprano nos quedará un único componente conexo gigante. De esto se desprende que las Redes Sociales son, casi siempre, grafos conexos.

16.1.4 Diámetro

El diámetro de un grafo es el máximo de todos los caminos mínimos. En otras palabras cuál es el máximo número de aristas que tenemos que visitar para llegar de un nodo del grafo a otro. Vamos a ver que las Redes Sociales tiene un diámetro muy bajo, lo cual da origen al fenómeno del mundo pequeño, los famosos 6 grados de separación. Otra propiedad sorprendente de las Redes Sociales es que a medida que las Redes Sociales crecen el diámetro en lugar de aumentar disminuye, esto no es algo lógico por lo que vamos a tener que tomarnos el tiempo para explicarlo mas adelante.

Otro valor asociado al diámetro es el camino mínimo promedio que surge del promedio de todos los caminos mínimos del grafo.

16.1.5 Coeficiente de Clustering

El coeficiente de clustering mide que tan conectados están los vecinos de un nodo. Sea k_i el grado del nodo entonces sus k_i vecinos pueden tener a lo sumo $1/2k_i(k_i - 1)$ aristas. Llamemos e_i a la cantidad de aristas entre los vecinos de i , el coeficiente de clustering para el nodo i es:

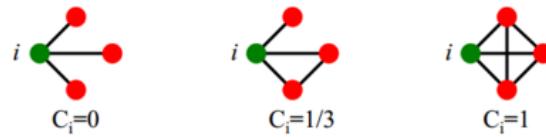


Figure 16.2: Coeficiente de Clustering

$$c_i = \frac{2e_i}{k_i(k_i - 1)} \quad (16.1)$$

Vamos a definir el coeficiente de clustering promedio de la Red como el promedio de todos los coeficientes de clustering:

$$C = \frac{\sum_i c_i}{N} \quad (16.2)$$

En una Red Social el coeficiente de clustering es en general un número alto, los vecinos de un nodo tienden a estar conectados entre sí precisamente por

tener en común su relación con ese nodo. A este fenómeno lo llamamos *clausura triangular* y lo vamos a estudiar mas adelante.

16.1.6 Distribución del Grado

La distribución del grado estudia de que forma se distribuye el grado de los nodos en una Red Social, es decir cuántos nodos tienen grado 1, cuántos nodos tienen grado 2, etc etc. Un fenómeno muy curioso es que en casi todas las Redes Sociales la distribución del grado se distribuye de acuerdo a una *ley de potencias* (power-law). En este tipo de distribución hay muchos nodos con grado bajo y pocos nodos con un grado muy alto, pero estos dominan la distribución por el fenómeno conocido como "long tail". El "long tail" significa que si bien la mayoría de los nodos tienen un grado bajo existen nodos con grados altísimos, completamente fuera de escala con respecto a los demás nodos de la Red Social.

16.2 Propiedades del Grafo de una Red Social

Hemos visto varias propiedades de una grafo y hemos comentado que tipo de valores podemos esperar en una Red Social. En este capítulo vamos a analizar el valor de estas propiedades para Facebook en base al trabajo de [Ugander]. El objetivo de esta sección es tomar idea de que tipo de valores toman las diferentes propiedades que hemos visto en una Red Social, mas adelante vamos a ver de que forma podemos interpretar estos valores mediante el uso de modelos de grafos.

16.2.1 Componente Conexo

En Facebook el 99.91% de los usuarios pertenecen a un único componente conexo. Si bien esto es esperable en grafos grandes el número es sorprendente ya que Facebook favorece la formación de pequeñas comunidades sin conexiones con el resto de la red, por ejemplo familias. Sin embargo es evidente que tarde o temprano estas pequeñas comunidades forman alguna amistad con un nodo en otra comunidad generando la conexión al resto de la Red.

16.2.2 Distribución del Grado

La distribución del grado en Facebook no sigue una ley de potencias, esto es muy fácil de explicar porque Facebook limita la cantidad máxima de contactos a 5000, por lo tanto no puede haber nodos con grado mayor a 5000 y esto evita que exista un "long tail". El fenómeno del Cisne Negro es imposible en Facebook ya que sabemos que incluso los usuarios mas populares no pueden tener mas de 5000 amistades.

Existe una anomalía cerca del número 20 porque Facebook alienta a los usuarios a formar nuevos contactos hasta que tienen al menos 20 contactos, esto se refleja en la distribución del grado haciendo que 20 sea un poco mas frecuente que 19 o 21.

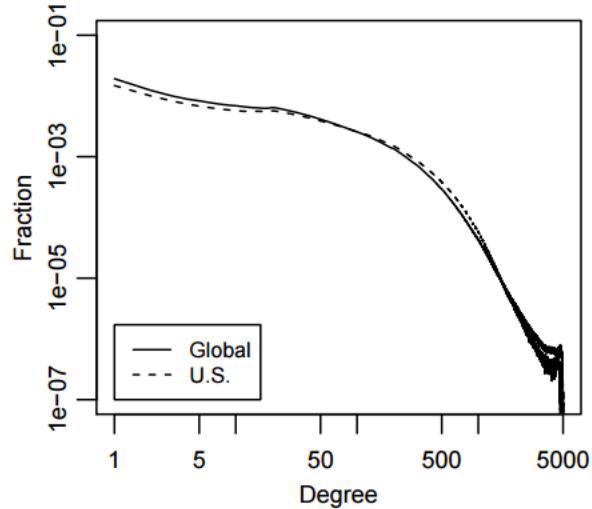


Figure 16.3: Distribución del Grado en Facebook

16.2.3 Diámetro y Camino Mínimo Promedio

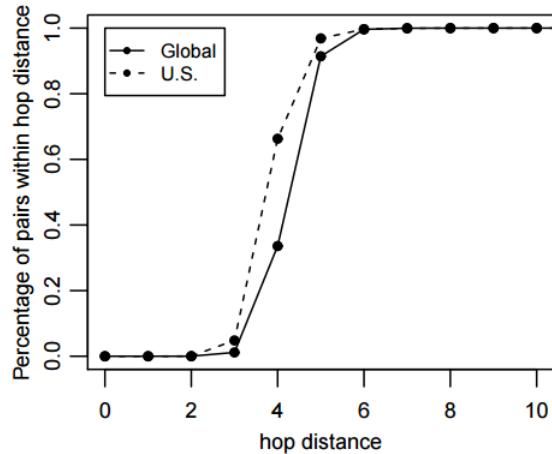


Figure 16.4: Cantidad de Linkes entre usuarios de Facebook

El camino mínimo promedio en Facebook es 4.7. El 99.6% de los usuarios pueden alcanzarse con un número de conexiones entre 2 y 6. Esto refleja que el fenómeno del mundo pequeño es muy fuerte en Facebook.

El diámetro de Facebook es 11, un número extremadamente pequeño considerando la cantidad de usuarios de la Red, esta es la cantidad máxima de links que tenemos que recorrer para llegar de un nodo a otro dentro del componente conexo gigante.

16.2.4 Coeficiente de Clustering

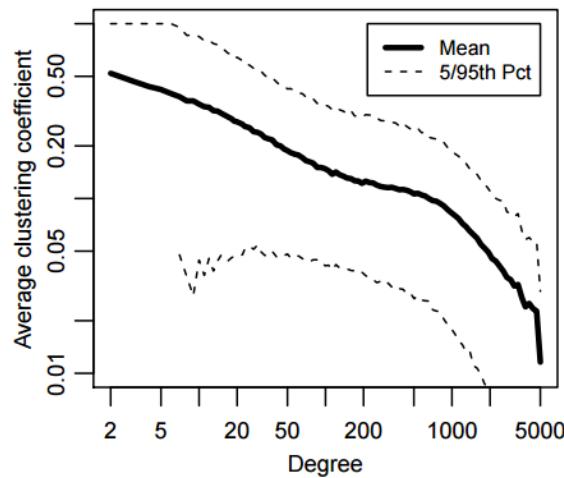


Figure 16.5: Coeficiente de Clustering en Facebook

El coeficiente de clustering promedio en Facebook es 0.14, este número es realmente enorme, es por ejemplo 5 veces mayor a otras redes sociales como MSN. En promedio si un usuario tiene n amigos un 14% de estos amigos estarán conectados entre sí.

El gráfico muestra que a medida que aumenta el grado de un nodo su coeficiente de clustering disminuye con una caída notable en el grado=5000. Esto quiere decir que los usuarios que tienen 5000 amistades probablemente no están usando la red con fines sociales sino con fines comerciales y por eso la notable disminución en su coeficiente de clustering. Sus amistades son clientes y no se conocen entre ellos.

16.3 Modelos de Grafos

Podemos realizar un resumen sobre lo que sabemos acerca de las características del grafo de una Red Social en la siguiente tabla:

Tenemos que entender ahora como detectar cuando el diámetro es alto o bajo o cuando el coeficiente de clustering es alto o bajo. Para hacer esto podemos

Diámetro	Muy pequeño
Coeficiente de Clustering	Muy alto
Componentes Conexos	1
Distribución del Grado	Power Law
Camino mínimo promedio	Bajo

Table 16.1: Características de una Red Social

comparar los valores de nuestra Red Social contra un grafo aleatorio. Los grafos aleatorios son uno de los varios modelos de grafos que vamos a estudiar, cada modelo va a ayudarnos a explicar diferentes características de una Red Social.

16.3.1 Modelo de Erdös-Renyi

El modelo de Erdös-Renyi es un grafo aleatorio con N nodos y probabilidad p para cada arista. A estos grafos los llamamos G_{np}

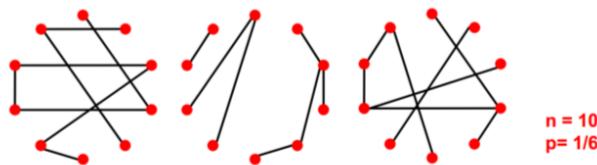


Figure 16.6: Modelo de Erdős-Renyi

Podemos armar este grafo de varias formas posibles. Una es considerar cada par de nodos posible del grafo y unirlos con probabilidad p , este método requiere enumerar todas las combinaciones de nodos lo cual no es eficiente. Un método idéntico pero mas eficiente es calcular la cantidad de aristas en el grafo como $pN(N - 1)/2$ y luego formar cada una de esas aristas tomando dos nodos al azar.

Diámetro

El diámetro en G_{np} está en el orden de $O(\log N)$ en el caso de Facebook con una muestra de 149 millones de usuarios el diámetro nos de 8 lo cual está muy cerca del valor real. Esto quiere decir que el diámetro de una Red Social no necesariamente obedece a características de la misma sino que puede ser, simplemente, una propiedad de un grafo aleatorio del mismo tamaño.

Distribución del Grado

El grado promedio de un nodo en G_{np} es $(n - 1)p$ lo cuál es lógico ya que $(n - 1)$ es el grado máximo y p es la probabilidad de que cada arista exista. La

distribución del grado en G_{np} sigue una distribución binomial con:

$$P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

La varianza es $\sigma^2 = p(1-p)(n-1)$.

La distribución tiene forma normal y no se parece en absoluto a la distribución de una ley de potencias que es la que esperamos en una Red Social, por lo tanto podemos decir que la formación de aristas en una Red Social no es aleatoria, si lo fuera la distribución sería normal.

Coeficiente de Clustering

Recordemos que el coeficiente de clustering se calculaba como:

$$c_i = \frac{2e_i}{k_i(k_i - 1)}$$

Podemos calcular e_i de la forma:

$$e_i = p \frac{k_i(k_i - 1)}{2}$$

Y entonces:

$$C_i = \frac{1}{k_i(k_i - 1)} \sum_{j \sim i} \frac{1}{k_j(k_j - 1)} = p = \frac{k}{n-1} \approx \frac{k}{N}$$

En definitiva el coeficiente de clustering es p que es un número muy pequeño. Esto tampoco coincide con lo que esperamos en una Red Social, por ejemplo el valor de p para Facebook es $p = 0.00000072$ que surge de analizar una muestra en donde tenemos 149 millones de nodos y 16000 millones de aristas. El total posible de aristas es (149 millones x 149 millones -1) y p es entonces (16000 millones / 149 millones al cuadrado). El coeficiente de clustering de Facebook sin embargo no es 0.00000072 sino que es 0.14 (!). Esto nos dice que en una Red Social el grado de conexión entre los amigos de un determinado nodo es realmente altísimo con respecto a lo que ocurriría simplemente por azar.

Del estudio de G_{np} podemos concluir que una Red Social no es random, el coeficiente de clustering denota una estructura local mucho mas alta que la que se esperaría por simple azar y la distribución del grado es completamente diferente. El siguiente modelo intentará explicar por qué la distribución del grado en las Redes Sociales sigue una ley de potencias.

16.3.2 Modelo de Barabasi-Albert

El modelo de Barabasi-Albert no genera las aristas del grafo de forma aleatoria sino que utiliza un modelo llamado *preferential attachment* la idea es que la probabilidad de agregar una arista en el grafo depende del grado de cada nodo, cuanto mayor el grado del nodo mas probable es que genere una nueva arista. Este fenómeno se puede explicar como "*the rich get richer*" ya que los nodos

con mas contactos son los que con mayor probabilidad van a generar nuevos contactos aumentando mas su grado y su probabilidad de generar contactos.

$$p_i = \frac{k_i}{\sum_j k_j}$$

El uso de Preferential attachment genera una distribución que es una ley de potencias con exponente $\alpha = 4$. El diámetro en este modelo es $O(\log \log N)$ que también es compatible con el diámetro de una Red Social.

Preferential Attachment genera la aparición de "Hubs", nodos que tienen muchos contactos y que mantienen la red unida, dado que hay muchos Hubs la red es resistente a fallas, la caída de miles de nodos, incluso Hubs no hace que la red deje de ser conexa, es decir que PA genera redes que resisten la desconexión de sus nodos manteniendo la conectividad de los nodos que quedan en la misma.

16.3.3 Modelo de Jackson-Rodgers

El modelo de Jackson-Rodgers es una combinación de G_{np} y el modelo de Barabasi-Albert. Vamos a usar un coeficiente α que será la probabilidad de generar un link al azar en la red. Luego $1 - \alpha$ es la probabilidad de generar un link siguiendo el proceso de Preferential attachment.

Es decir que si $\alpha = 0$ el modelo es igual a preferential attachment y si $\alpha = 1$ el modelo es G_{np}

Empíricamente se puede mostrar que en la mayoría de las Redes Sociales puede modelarse la formación de links de forma muy precisa para un cierto parámetro α que depende de la red. Por ejemplo para la web tenemos $\alpha = 0.36$ es decir que $2/3$ de los links en la web responden a preferential attachment y $1/3$ son aleatorios.

El problema del modelo es la necesidad de encontrar el valor de α que se tiene que buscar mediante una regresión lineal.

16.4 El Mundo Pequeño

El fenómeno del mundo pequeño surge en 1967 a partir de un famoso estudio realizado por Stanley Milgram. Milgram eligió a 300 personas en los estados de Omaha, Nebraska, Wichita y Kansas y les pidió que le hicieran llegar una carta a una persona de Boston de la cual solo se conocía su nombre. Estas personas al no conocer al destinatario tenían que enviar la carta a alguien que supusieran los acercaría al mismo. El objetivo del experimento era ver cuantos pasos tomaría llegar desde una persona al azar hasta cualquier otra usando el correo.

El resultado fue el que todos conocemos: en promedio fueron necesarios 6.2 pasos para llegar al destinatario en Boston. De aquí surgen los famosos 6 grados de separación.

El fenómeno se hizo popular cuando aparecieron muchos sitios como el "Oracle of Bacon" anunciando que era posible conectar a cualquier actor con Kevin

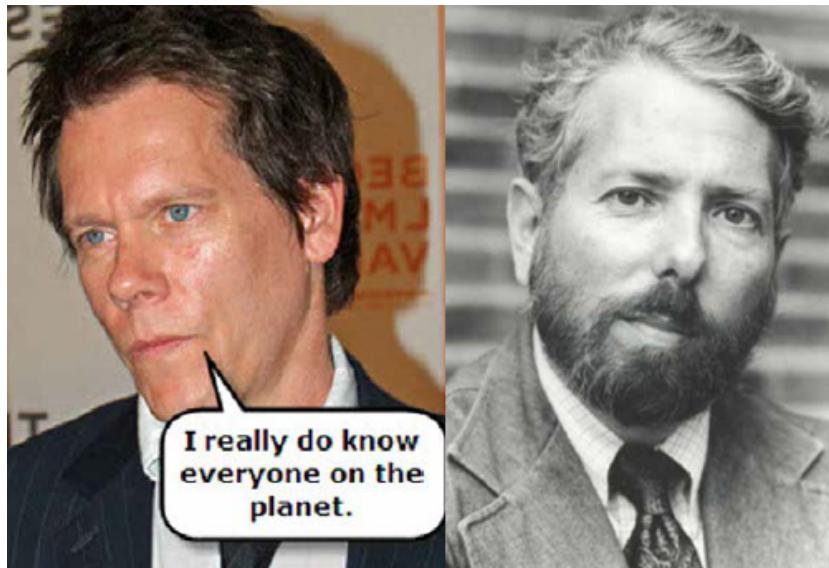


Figure 16.7: Kevin Bacon y Stanley Milgram

Bacon en 6 o menos links. Esto no sólo es sorprendente sino que también es cierto. A la cantidad de links que unen a un actor con Kevin Bacon se la llama "número Bacon", el máximo es 10 y es realmente terriblemente difícil de encontrar. Para la mayoría de los actores es posible encontrar una conexión en menos de 6 links. Por ejemplo, el ganador del Oscar Guillermo Francella tiene número Bacon 3. Francella trabajó en "Camarero Nocturno en Mar del Plata" junto con Rodolfo Ranni, Ranni trabajó en "El Sekuestro" junto con José Ramón Rosario y este último actuó en Mystic River junto a Kevin Bacon.

En un grafo aleatorio G_{np} tenemos presente el fenómeno del mundo pequeño pero el coeficiente de clustering promedio es muy bajo. Si organizamos los nodos en una "grilla" en donde cada nodo se conecta con k vecinos tenemos un coeficiente de clustering alto pero no tenemos el fenómeno del mundo pequeño. El problema es como encontrar un modelo que nos de ambas cosas.

16.4.1 Modelo de Watts Strogratz

En este modelo vamos a partir de una grilla regular, por ejemplo en donde todo nodo está conectado a sus dos vecinos más cercanos. Esta grilla tiene clustering alto pero también diámetro alto, es decir que no responde al fenómeno del mundo pequeño. Por cada arista con probabilidad q vamos a cambiar uno de sus nodos por cualquier otro del grafo.

Si usamos $q = 0$ tenemos la grilla regular con alto clustering y alto diámetro. Si usamos $q = 1$ tenemos G_{np} con bajo clustering y bajo diámetro. El objetivo

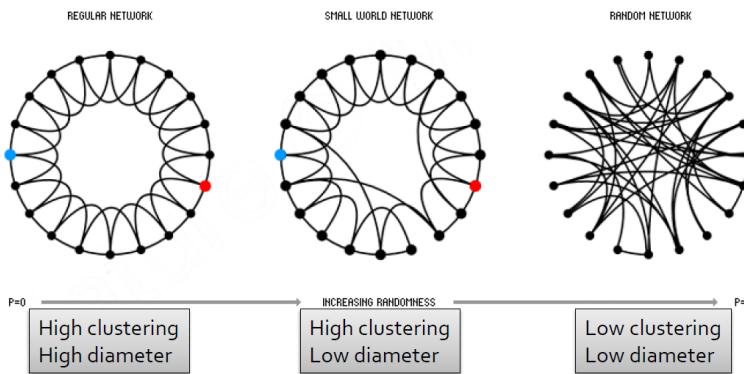


Figure 16.8: Modelo de Watts-Strogatz

es encontrar q tal que tengamos alto clustering y bajo diámetro como en las Redes Sociales.

Se demuestra que con valores bajos de q se logra un grafo con alto clustering y bajo diámetro es decir que agregar unos pocos links al azar alcanzan para que el modelo de Watts-Strogatz se convierta en una Red Social. Lamentablemente el modelo no tiene la distribución del grado que esperamos en una Red Social ya que la misma no sigue una ley de potencias.

16.5 La Clausura Triangular

La clausura triangular es el mecanismo que origina el coeficiente de clustering alto en las Redes sociales y puede enunciarse de forma muy simple: "si A conoce a B y C entonces tarde o temprano B conoce a C"

La existencia de una cantidad inusualmente alta de triángulos automáticamente implica un coeficiente de clustering alto, contra triángulos dentro de un grafo es entonces una de las formas mas efectivas de determinar si el grafo responde a las características de una Red Social.

16.5.1 Contando Triángulos en un Grafo

El método trivial consiste en tomar cada nodo y por cada nodo tomar su vecinos y verificar si para cada par de vecinos existe un link entre ellos. Este método nos daría en nuestro ejemplo el total de 7 triángulos: 1-6-7,1-7-9,2-3-5,2-3-4,2-4-5,3-4-5,6-7-8. El problema del método es que es ineficiente y es inviable en grafos grandes. Por eso vamos a intentar un método aproximado que es mucho mas eficiente [Charalampos].

Observemos la matriz de adyacencias de nuestro grafo:

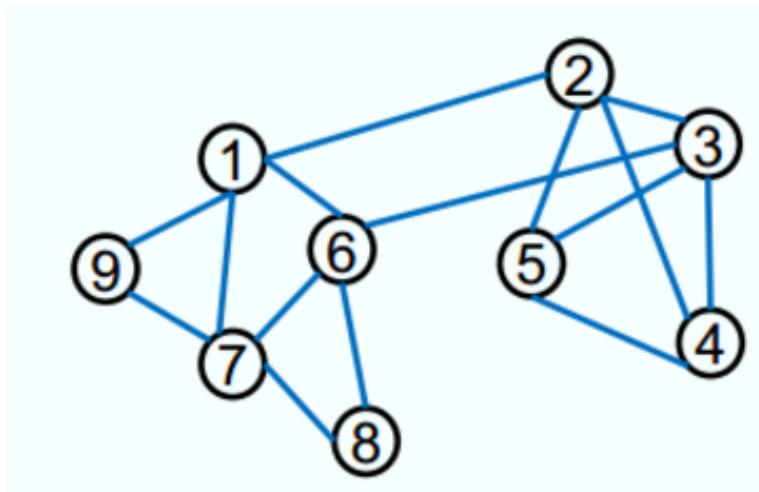


Figure 16.9: Contando triángulos en una Red Social

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Calculamos M^2

$$M^2 = \begin{pmatrix} 4 & 0 & 2 & 1 & 1 & 1 & 2 & 2 & 1 \\ 0 & 4 & 2 & 2 & 2 & 2 & 1 & 0 & 1 \\ 2 & 2 & 4 & 2 & 2 & 0 & 1 & 1 & 0 \\ 1 & 2 & 2 & 3 & 2 & 1 & 0 & 0 & 0 \\ 1 & 2 & 2 & 2 & 3 & 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 1 & 1 & 4 & 2 & 1 & 2 \\ 2 & 1 & 1 & 0 & 0 & 2 & 4 & 1 & 1 \\ 2 & 0 & 1 & 0 & 0 & 1 & 1 & 2 & 1 \\ 1 & 1 & 0 & 0 & 0 & 2 & 1 & 1 & 2 \end{pmatrix}$$

El elemento ij de M^2 representa los caminos de longitud 2 que empiezan en i y terminan en j . Por ejemplo no hay ningún camino de longitud 2 que empiece en 1 y termine en 2. Hay 4 caminos de longitud 2 que empiezan y terminan en 1 (1-9-1,1-6-1,1-2-1,1-7-1). La diagonal de la matriz es el grado de cada vértice.

$$M^3 = \begin{pmatrix} 4 & 9 & 3 & 3 & 3 & 10 & 8 & 3 & 6 \\ 8 & 6 & 10 & 8 & 8 & 3 & 3 & 3 & 1 \\ 3 & 10 & 6 & 8 & 8 & 8 & 3 & 1 & 3 \\ 3 & 8 & 8 & 6 & 7 & 3 & 2 & 1 & 1 \\ 3 & 8 & 8 & 7 & 6 & 3 & 2 & 1 & 1 \\ 10 & 3 & 9 & 3 & 3 & 4 & 8 & 6 & 3 \\ 8 & 3 & 3 & 2 & 2 & 8 & 6 & 6 & 6 \\ 3 & 3 & 1 & 1 & 1 & 6 & 6 & 2 & 3 \\ 6 & 1 & 3 & 1 & 1 & 3 & 6 & 3 & 2 \end{pmatrix}$$

De la misma forma M^3 representa los caminos de longitud 3. La diagonal de la matriz representa ahora todos los caminos de longitud 3 que empiezan y terminan en el mismo nodo. Para el nodo 1 por ejemplo son 4: (1-7-9-1, 1-9-7-1, 1-6-7-1, 1-7-6-1). Por lo tanto la diagonal dividido 2 es la cantidad de triángulos en las que participa cada nodo. Esto quiere decir que la *traza* de la matriz nos da el total de triángulos multiplicado por 6, por 2 por el motivo anterior y por 3 porque cada triángulo tiene 3 nodos participantes entonces para (1-7-9) vamos a contar 1 para el nodo 1, el 7 y el 9. En nuestro ejemplo la traza de la matriz es 42 que dividido por 6 nos da 7 que es el total de triángulos en el grafo.

Pero por supuesto no podemos pretender elevar una matriz que ni siquiera podemos almacenar al cubo. Afortunadamente esto no hace falta. Sabemos que si λ es un autovalor de A entonces λ^n es un autovalor de A^n y sabemos que la suma de los autovalores es igual a la traza de la matriz. Por lo tanto podemos estimar la cantidad de triángulos en el grafo como:

$$\frac{1}{6} \sum_{i=1}^n \lambda_i^3 \quad (16.3)$$

Los autvalores de la matriz se pueden obtener de forma muy eficiente con el método de Lanczos y ni siquiera nos hacen falta todos, podemos estimar la cantidad de triángulos a partir de los k autovalores de mayor valor absoluto de la matriz con un error muy pequeño. El método de Lanczos está basado en el power method con el que calculamos PageRank, recordemos que el power method nos da el autovalor principal de la matriz.

Como un múltiplo de un autovector también es un autovector vamos a normalizar el vector resultado de cada iteración del power method de forma tal que tenga norma 1. De esta forma podemos obtener el autovector asociado al autovalor principal cuya norma es 1. El autovalor asociado se puede calcular simplemente mediante:

$$\lambda = x^t A x \quad (16.4)$$

Donde x es el autovector principal. Esto surge de despejar λ en $Ax = \lambda x$.

Una vez que se obtiene el autovector asociado al autovalor principal podemos neutralizar el efecto del mismo en la matriz para encontrar el segundo autovector. Para eso usamos:

$$A^* = A - \lambda xx^t \quad (16.5)$$

Aplicando el power method a la nueva matriz A^* obtenemos el segundo autovector. Repitiendo este método obtenemos uno a uno los autovectores y autovalores de una matriz de forma iterativa.

En nuestro ejemplo los autovalores son:

$$\lambda = \begin{pmatrix} -11.32 \\ -3.80 \\ -2.12 \\ -1 \\ -0.17 \\ -0.01 \\ 0.51 \\ 16.80 \\ 43.13 \end{pmatrix}$$

La suma de todos da 42 y por supuesto $42/7=6$ que es la cantidad de triángulos. Observemos que el autovalor principal es $\lambda_1 = 43.13$ es decir que podríamos estimar la cantidad de triángulos calculando simplemente $\lambda_1/6$ con buen resultado. El Power-Method nos da este autovalor automáticamente.

16.5.2 Triángulos Locales

Los triángulos locales son la cantidad de triángulos en los cuáles participa un cierto nodo. Esto nos da una medida de que tan sociable es el nodo en la Red. Podemos contar los triángulos locales mediante un método similar al anterior usando los autovalores y autovectores de la matriz de adyacencias del grafo.

$$\nabla_i = \frac{\sum_j \lambda_j^3 u_{ij}^2}{2} \quad (16.6)$$

Donde u_{ij} es la i-ésima entrada del j-ésimo autovector.

Si graficamos la cantidad de triángulos en los que participa un nodo lo que tenemos es, oh-sorpresa! una ley de potencias. En escala logarítmica tenemos una recta casi perfecta. Esto quiere decir que la mayoría de los nodos participa en una cantidad de triángulos baja pero hay nodos que participan en una cantidad enorme de triángulos. Es muy interesante detectar cuales son esos nodos. En Twitter, por ejemplo Barack Obama participa de una cantidad descomunal de triángulos lo cual refleja que sus seguidores son altamente sociales. Esto no es tan evidente en los seguidores de McCain, quizás esto explique algún resultado electoral.

Es interesante analizar la relación entre el grado de un nodo y la cantidad de triángulos en los que participa el mismo. Esto sirve para detectar cuentas anómalas. En Twitter la mayoría de las cuentas anómalas tiene una cantidad

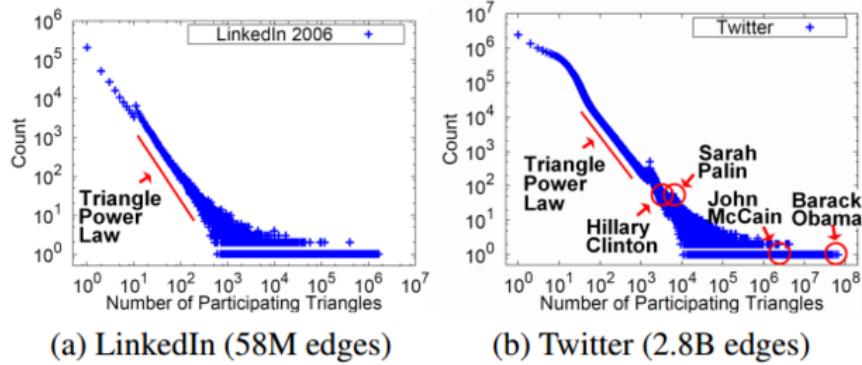


Figure 16.10: Triángulos en una red Social

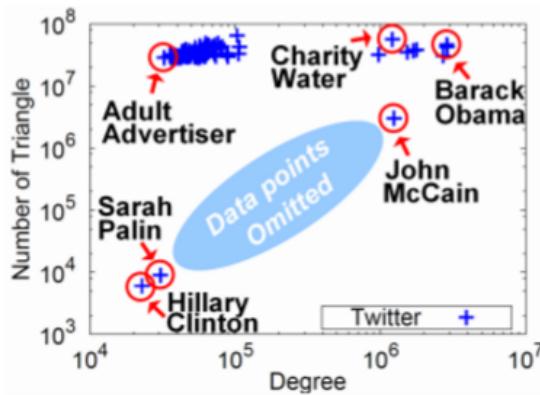


Figure 16.11: Grado vs Cantidad de triángulos

de triángulos inusualmente alta considerando el grado del nodo. Para las celebridades el grado es alto pero la cantidad de triángulos es menor ya que sus seguidores no se conocen entre sí.

16.5.3 EE Eigenplots

Se puede realizar un análisis del espectro de la matriz de adyacencias de una Red Social ploteando los autovectores de la matriz de adyacencias entre sí. [Kang-Meeder-Faloutsos]. Esto permite encontrar todo tipo de estructuras como flares, cliques y otras.

Podemos ver plots en donde cada eje es un autovector de la matriz de adya-

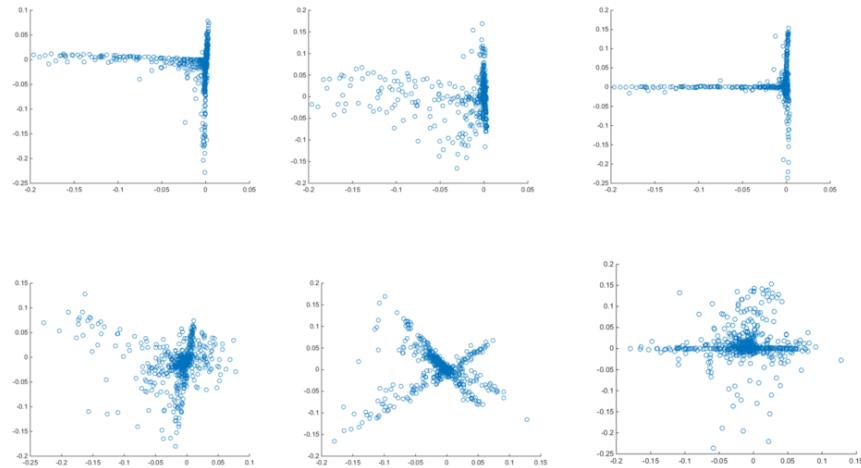


Figure 16.12: Eigenplots en una Red Social

cencias de la Red Social, estos plots nos dan una representación de la topología de la Red, cada uno tiene información que podemos aprovechar para detectar comunidades, estructuras bipartitas, grupos anómalos y otras características. Lo único que tenemos que hacer es identificar cuáles son los nodos que participan de las estructuras que nos llamen la atención de los distintos plots. En el primer plot tenemos dos "direcciones", esto nos da una clasificación de la red en dos grupos y es interesante averiguar cual es el factor que separa los dos grupos.

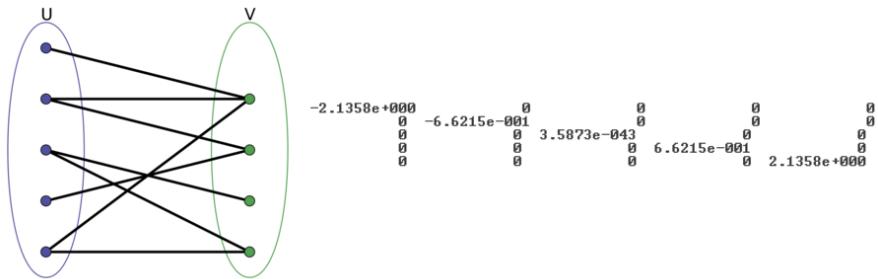


Figure 16.13: Grafos Bipartitos

Los autovalores de la matriz de adyacencias de un grafo grande son valiosos también para aproximar otros problemas difíciles. Si un grafo es bipartito entonces sus autovalores son simétricos, por ejemplo $(3, 2, 1, -2, -3)$. Esto sirve para encontrar dentro de un grafo grande estructuras bipartitas es decir en donde tenemos dos grupos que interactúan entre sí pero no existe relación alguna entre

los miembros de cada grupo.

16.6 Centralidad

En esta sección vamos a estudiar el concepto de *centralidad* en Redes Sociales, lo que queremos es una medida de la importancia que cada nodo tiene en la Red. Existen muchas formas de medir la centralidad y cada métrica es útil para diferentes tipos de Redes, vamos a analizar las diferentes definiciones que podemos tener para calcular la centralidad de un nodo y sus aplicaciones.

16.6.1 Grado

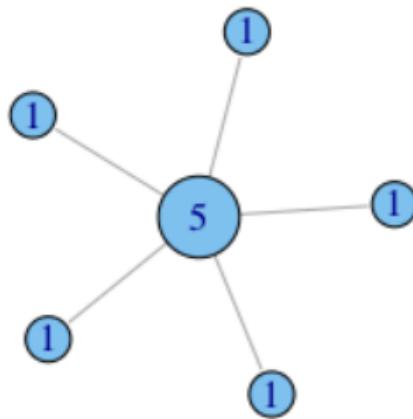


Figure 16.14: Centralidad: Grado

El grado de un nodo es la medida mas simple de centralidad, a mayor cantidad de vecinos (amigos) mas importante es el nodo. En esta métrica suponemos que no importa quienes son los vecinos de un nodo sino simplemente cuántos son.

Para poder comparar la centralidad de dos nodos es recomendable normalizar el grado dividiendo por el grado máximo que sabemos es $N - 1$ esto permite comparar la centralidad de dos nodos aunque estén en redes diferentes.

El grado no captura la importancia que tiene un nodo como nexo de dos o mas comunidades (*brokerage*). En la figura 16.16 vemos que tanto *Y* como *X* tienen grado 2, sin embargo es lógico pensar que si tenemos una red de comercio el nodo *X* es mucho mas importante ya que nos permite comunicar a

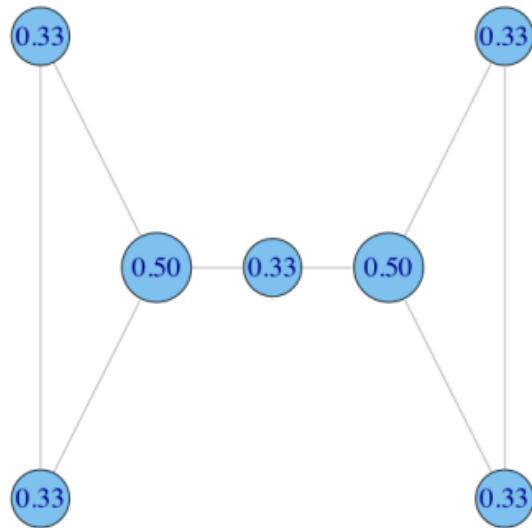


Figure 16.15: Centralidad: Grado Normalizado

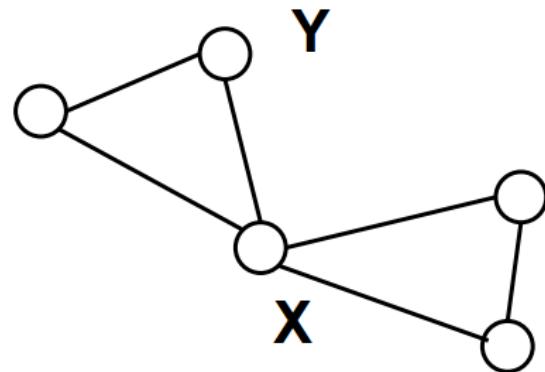


Figure 16.16: Brokerage

los nodos que están a cada lado de él. Para medir esto usaremos el concepto de *betweenness*

16.6.2 Betweenness

La definición de betweenness es: "la cantidad de caminos mínimos que pasan por un nodo o arista". Es decir que puede usarse tanto para calcular la importancia de los nodos como de las aristas.

La definición formal es:

$$C_B(i) = \sum_{j < k} g_{jk}(i)/g_{jk} \quad (16.7)$$

En donde g_{jk} es la cantidad de caminos mínimos que conectan j y k y $g_{jk}(i)$ es la cantidad de caminos mínimos entre j y k de los que participa i . En general podemos normalizar el betweenness dividiendo por $(n - 1)(n - 2)/2$ que es la cantidad de pares de vértices que existen excluyendo al vértice mismo. La normalización es opcional.

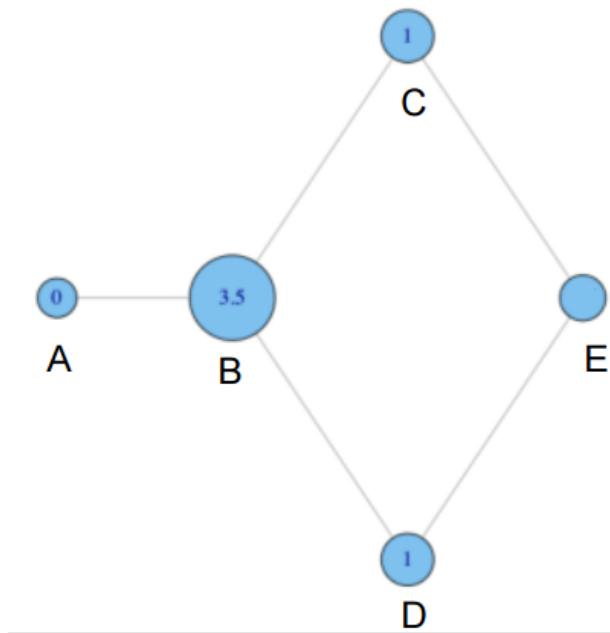


Figure 16.17: Betweenness

Veamos los caminos mínimos para nuestro ejemplo:

$$A-B = A-B$$

$$A-C = A-B-C \text{ (B suma 1 punto)}$$

$$A-D = A-B-D \text{ (B suma 1 punto)}$$

$$A-E = A-B-C-E \text{ o } A-B-D-E \text{ (B suman 1 punto, C y D suman 1/2 punto)}$$

$$B-C = B-C$$

$$B-D = B-D$$

$$B-E = B-C-E \text{ o } B-D-E \text{ (C y D suman } 1/2 \text{ punto)}$$

$$C-D = C-B-D \text{ o } C-E-D \text{ (B y E suman } 1/2 \text{ punto)}$$

$$C-E = C-E$$

$$E-D = E-D$$

Es decir que el betweenness de "E" que falta calcular es 0.5 ya que participa de "medio" camino mínimo entre C y D ya que hay dos caminos mínimos entre C y D y E solo está en el medio de uno de ellos.

En algunos casos no queremos medir la cantidad de amigos que tiene un nodo ni que tan "en medio" se encuentra el nodo sino que queremos medir que tan cerca esta el nodo del centro de la red. Esto lo vamos a medir mediante la centralidad que llamaremos *closeness*

16.6.3 Closeness

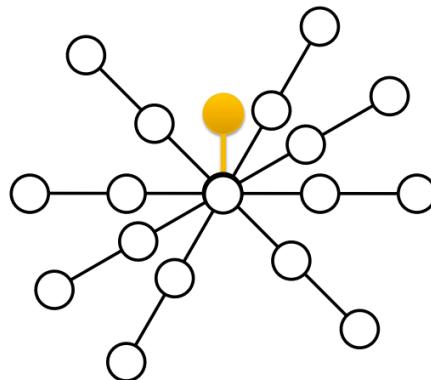


Figure 16.18: Closeness

En la figura 16.18 el nodo amarillo solo tiene grado 1 y no tiene un betweenness alto sin embargo esta cerca del "centro" del grafo. El concepto de *closeness* está basado en el promedio de los caminos mínimos entre el nodo y todos los demás nodos de la red. Como este valor nos da mas chico cuanto mas central es el nodo usamos el recíproco es decir 1 dividido el promedio de los caminos mínimos, de esta forma la centralidad nos da mayor cuanto mas cerca del centro de la red se encuentre el nodo.

$$C_c(i) = [\sum_{j=1}^N d(i,j)]^{-1} \quad (16.8)$$

Opcionalmente podemos normalizar dividiendo por $(N - 1)$.

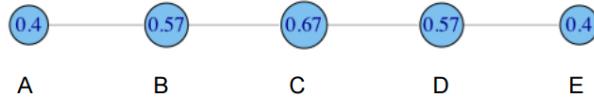


Figure 16.19: Closeness, ejemplo

En el ejemplo usamos la versión de closeness normalizada. Los caminos mínimos desde A tienen longitud 1,2,3 y 4 respectivamente hacia B,C,D y E por lo tanto el valor de centralidad closeness para A es: $[1 + 2 + 3 + 4/4]^{-1} = 0.4$

16.6.4 Eigenvector Centrality y PageRank

El concepto de EigenVector Centrality es que la centralidad de un nodo depende de la centralidad de los nodos que apuntan hacia dicho nodo. Es decir la misma idea que vimos al estudiar PageRank. EigenVector Centrality es PageRank sin teletransportación. No vamos a entrar en detalles sobre estas métricas ya que las hemos analizado extensamente en el capítulo sobre PageRank.

16.6.5 Centralidad de Bonacich

La centralidad de Bonacich es una variante de la Eigenvector centrality. Se define de la siguiente forma:

$$c_i(\beta) = \sum_j (\alpha + \beta c_j) A_{ji} \quad (16.9)$$

$$c(\beta) = \alpha(I - \beta A)^{-1} A \mathbf{1} \quad (16.10)$$

α es una constante de normalización. β determina que tan importante es la centralidad de los vecinos. A_{ij} es la matriz de adyacencias. Es decir que solamente sumamos $\alpha + \beta c_j$ para aquellos nodos j que llevan a i .

Cuando β es un número chico solo los vecinos inmediatos son importantes. Cuando β es un número grande el efecto global de la red es el factor mas importante.

Notemos que si $\beta = 0$ nos queda $c_i = \sum_j \alpha A_{ij}$ que es la definición del grado del nodo.

Cuando $\beta > 0$ los nodos tienen mayor centralidad cuando tienen aristas hacia otros nodos con mucha centralidad. Cuando $\beta < 0$ los nodos tienen mayor centralidad cuando tienen aristas hacia nodos con poca centralidad.

En la figura 16.20 vemos como con $\beta = 0.25$ el nodo de mayor centralidad es el del medio, pero cambiando el signo de β los nodos que están a los lados del del medio son mas importantes, esto es porque tienen links a los nodos de las puntas que tienen baja centralidad mientras que el nodo central no tiene links a estos nodos.

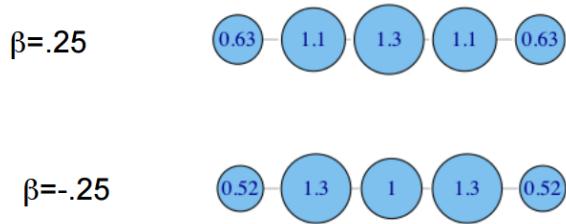


Figure 16.20: Centralidad de Bonacich

16.6.6 Centralidad de la Red (Freeman)

La centralidad de Freeman se usa para medir el grado de centralidad de la Red en su totalidad. La definición es:

$$C_D = \frac{\sum_{i=1}^g [C_D(n^*) - C_D(i)]}{(N-1)(N-2)} \quad (16.11)$$

En donde $C_D(i)$ es la centralidad del nodo i de acuerdo a alguna métrica y $C_D(n^*)$ es la centralidad máxima de cualquier nodo de la red.

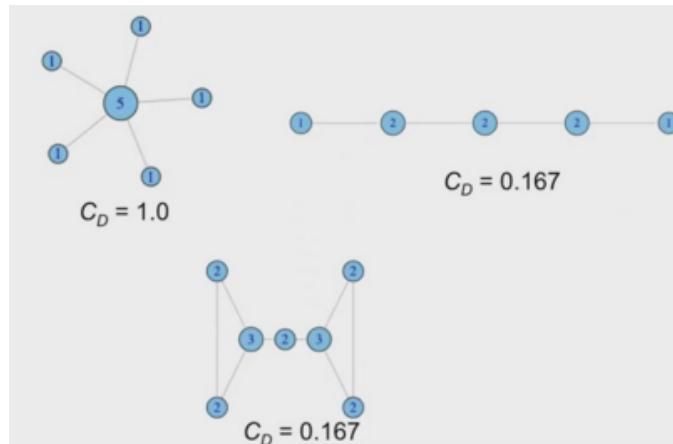


Figure 16.21: Centralidad de Freeman

La centralidad de Freeman es un número entre 0 y 1, una red con centralidad 1 indica que hay un nodo que es el que tiene máxima centralidad y todos los demás son mínimos, es decir una topología tipo estrella. Una centralidad cercana a 0 indica una red cuya topología está fuertemente des-centralizada y todos los nodos tienen centralidades muy similares. En general esto evidencia una estructura más caótica.

16.7 Detección de Comunidades

16.7.1 Uniones Fuertes y Débiles

Podemos caracterizar las aristas de una red según la *fuerza* de su uniones. En general una unión es débil cuando al removerla el camino mínimo entre los nodos que la arista unía se incrementa. Definimos el "span" de una arista como la distancia entre los nodos si removemos la arista. Un "puente local" es una arista cuyos vecinos no tienen vecinos en común.

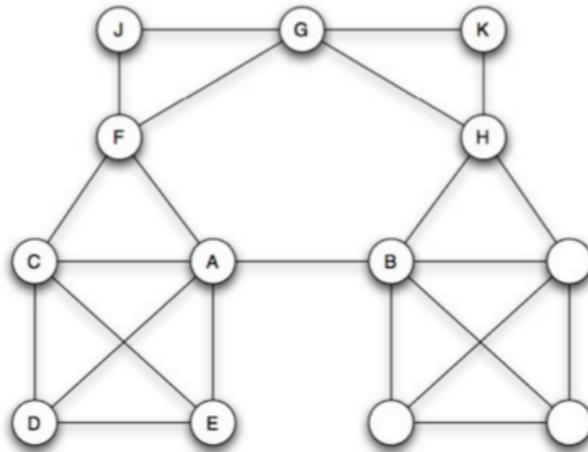


Figure 16.22: Uniones débiles

La arista A-B es un puente local ya que A y B no tienen vecinos en común y el span de esta arista es 4 ya que si removemos la arista el camino mínimo entre A y B pasa a ser 4.

Un estudio realizado por Granovetter comprobó que las uniones débiles son muy importantes en las Redes ya que son aquellas que permiten "descubrir" cosas. Por ejemplo si una persona está buscando un trabajo, aquellas amistades que lo conectan con grupos de interés diferentes al círculo cercano de la persona son los que mayor probabilidad tienen de referenciar a la persona para un nuevo trabajo.

La conclusión es que una Red Social es un conjunto de comunidades unidas entre sí por uniones débiles. Detectar estas comunidades es algo muy importante en el estudio de Redes Sociales.

16.7.2 Modularidad

La función de modularidad [Clauset-Newman-Moore] permite evaluar que tan modular es una red separada en un cierto conjunto de comunidades. La función supone la existencia de una función $\delta(v_1, v_2)$ que vale 1 si los vértices v_1 y v_2 están en la misma comunidad y cero en caso contrario. La definición de modularidad es la siguiente:

$$Q = \frac{1}{2m} \sum_{vw} [A_{vw} - \frac{K_v K_w}{2m}] \delta(c_v, c_w) \quad (16.12)$$

Donde A_{vw} es la matriz de adyacencias. K_v y K_w son el grado de los vértices. m es la cantidad total de aristas en el grafo y δ es la función que ya hemos explicado. Para redes completamente aleatorias $Q = 0$.

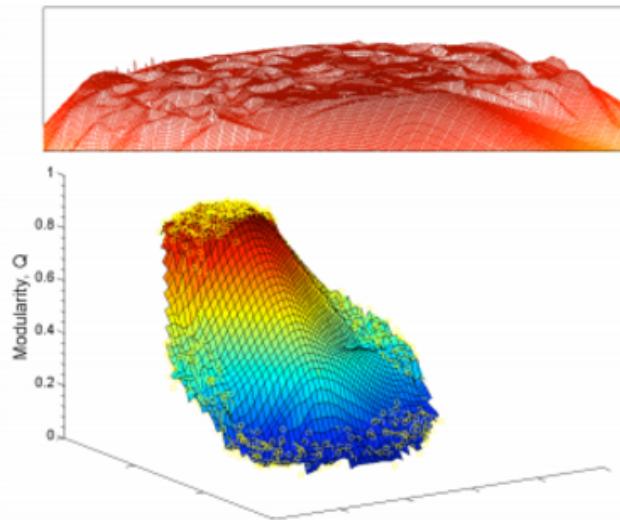


Figure 16.23: Función Modularidad

La función de modularidad tiene una forma bastante particular [Good-Montjoye-Clauset] que es propicia a ser optimizada por un algoritmo del estilo Hill-Climbing, sin embargo las variantes por Hill Climbing pueden ser ineficientes, existe una variante greedy para optimizar la modularidad que es la que actualmente se usa.

Algoritmo de Louvain

El algoritmo de Louvain [Louvain] es muy simple: inicialmente cada nodo pertenece a su propia comunidad, es decir que hay tantas comunidades como nodos y la función $\delta(x, y)$ vale 0 para todos los pares de nodos. La modularidad inicial es entonces $1/2m$. Luego por cada nodo de la red buscamos asignarlo a la

comunidad vecina que maximice la modularidad. Una vez que hemos recorrido todos los nodos asignando a cada uno a la comunidad que corresponde vamos a considerar a cada comunidad como si fuera un nodo y repetimos el proceso. Cuando consideramos a cada comunidad un nuevo nodo los links entre nodos dentro de esta comunidad se consideran links del nodo a si mismo.

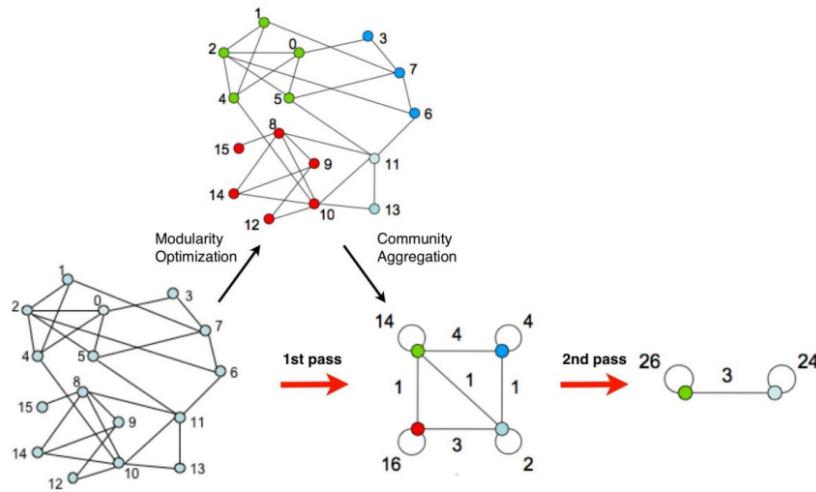


Figure 16.24: Función Modularidad

En el ejemplo vemos como el primer paso detecta 4 comunidades diferentes: [0,1,2,4,5] [3,7,6] [8,9,10,12,14,15] [11 13]. En la segunda etapa construimos el pequeño grafo que tiene solo 4 nodos, uno por cada comunidad detectada en el paso anterior. En este paso unimos la comunidad verde con la azul y la roja cola celeste. Finalmente quedan detectadas dos comunidades en el grafo una con los nodos [0,1,2,3,4,5,6,7] y la otra con [8,9,19,11,12,13,14,15] finalmente el algoritmo detecta que unir estas dos comunidades no aumenta la modularidad por lo que se detiene con un resultado final de dos comunidades.

En redes que son muy grandes el algoritmo de Louvain tiende a detectar comunidades grandes ya que suele pasar que mergear dos comunidades aumente la modularidad, es decir que la *resolución* del algoritmo depende del tamaño de la red y en redes muy grandes el algoritmo no puede detectar comunidades pequeñas.

16.7.3 Algoritmo de Girwan-Newman

El algoritmo de Girwan-Newman realiza una descomposición de la red usando el cálculo de betweenness para cada arista. En cada paso el algoritmo elimina la arista con mayor betweenness. El algoritmo se detiene cuando se ha llegado a

la cantidad de comunidades que estamos buscando, o con algún otro criterio.

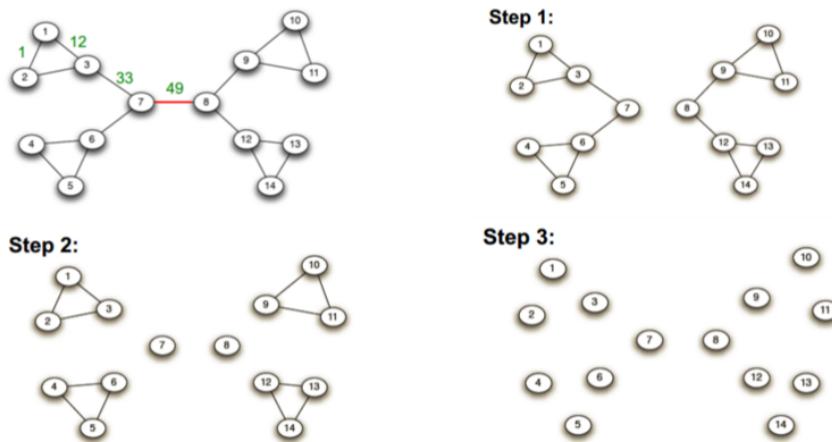


Figure 16.25: Algoritmo de Girwan-Newman

En el ejemplo la primer arista a eliminar es la que tiene betweenness = 49, esto nos divide la red en dos comunidades. En el segundo paso nos quedan 6 comunidades, 4 de 3 nodos cada una y dos nodos sueltos. Finalmente podemos seguir hasta que solo quedan comunidades de un único nodo lo cual en general no tiene mucho sentido.

16.7.4 Clustering Espectral

El algoritmo de Clustering Espectral puede aplicarse directamente sobre la matriz de adyacencias de la Red de forma tal de obtener una cantidad de comunidades que tenemos que conocer con anticipación. Este algoritmo ya lo hemos desarrollado en el capítulo sobre clustering por lo que no vamos a volver a entrar en detalles.

16.7.5 Infomap

Infomap [Rosvall-Bergstrom] es un algoritmo para detectar comunidades en redes sociales basado en los principios de la compresión de datos y la teoría de la información.

Dada una red dirigida o no dirigida podemos generar un random-walk de longitud arbitraria, que suponemos es infinito, si la red es dirigida agregamos teletransportación con $\beta = 0.85$ para evitar quedar atrapados sin salida en algún ciclo de la red. En este random walk podemos contar la frecuencia con la que visitamos cada nodo y por lo tanto la probabilidad de visitar el mismo, con esta probabilidad podríamos calcular la longitud en bits de cada nodo en base a la

fórmula que conocemos basada en la entropía: $L_i = -\log_2(P_i)$. Esto nos daría una forma de comprimir un random-walk en una red en donde no hay clusters.

En una Red en donde existen clusters o comunidades es posible que el random-walk quede ciclando en estas comunidades durante un cierto tiempo hasta eventualmente encontrar una salida, los clusters son entonces una forma de *contexto* que podemos usar para comprimir mejor: calcularemos las probabilidades de los nodos dentro de su cluster y no de forma global.

Supongamos una red en donde ya conocemos cuáles son las comunidades (clusters), vamos a usar un árbol de huffman o tabla de probabilidades para indicar cuál es el cluster en cual empezamos, una vez dentro del cluster vamos a tener una tabla de huffman o tabla de probabilidades para cada nodo dentro del cluster mas un elemento extra que usaremos para decir que salimos del cluster.

Un random-walk arbitrario puede codificarse entonces de la forma: entrar-cluster-A,A1,A3,A2,A4,A1,A2,salir-cluster-A, entrar-cluster-C, C2,C1,C3,C2,C1,C3, salir-cluster-C... etc. Notemos que cuando encontramos el símbolo correspondiente a salir de un cluster tenemos que cambiar de tabla o árbol a aquella que nos indicará a que cluster ingresar a continuación.

De acuerdo a lo que aprendimos en compresión de datos sobre el uso de contextos dividir la red en clusters nos permite comprimir mas ya que podemos usar códigos compactos dentro de cada cluster en lugar de tener que compartir todos los códigos juntos en un único árbol para toda la red.

El objetivo de Infomap es encontrar cuál es la partición de nodos en comunidades que minimiza la longitud en bits necesaria para codificar los random-walks dentro de la misma. Esto se puede expresar mediante la ecuación denominada "Map Equation"

Definition 16.1 (Map Equation)

$$L(M) = q \curvearrowright H(\mathbb{Q}) + \sum_{i=1}^m p^i \circlearrowleft H(\mathbb{P}^i) \quad (16.13)$$

La ecuación es importante pero en realidad no es necesaria, simplemente podemos calcular la longitud que tomaría nuestro random-walk codificándolo en base a las probabilidades que podemos obtener del random-walk mismo. Veamos el siguiente ejemplo:

`RW = A,B,C,B,A,C,D,E,F,D,F,D,C,B,A,C,C,B`

Supongamos en primer lugar que usamos un único cluster. Calculamos entonces la probabilidad de cada nodo:

En total necesitamos 43.9392 bits para representar al random-walk. Veamos qué pasa ahora si decidimos particionar la red en dos clusters: un cluster con {A,B,C} y otro con {D,E,F}. Nuestro random walk es ahora:

`RW = CL1,A,B,C,B,A,C,EXIT1,CL2,D,E,F,D,F,D,EXIT2,CL1,C,B,A,C,C,B,EXIT1`

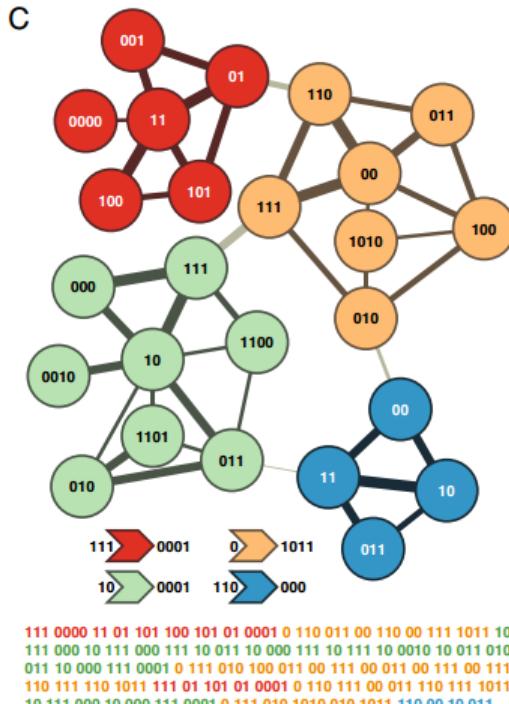


Figure 16.26: Ejemplo de codificación de un Random-Walk en una red. Tenemos cuatro clusters los códigos de cada nodo dentro de cada cluster responden a la probabilidad de los mismos. Debajo del grafo vemos los códigos usados para ingresar y salir de cada cluster. Notar que el código de salida comparte el mismo árbol de codificación que los nodos del cluster mientras que los códigos de ingreso a cada cluster constituyen un segundo árbol independiente

Node	Prob
A	3/18
B	4/18
C	5/18
D	3/18
E	1/18
F	2/18

Table 16.2: Ejemplo sin clusters

Nuestras probabilidades ahora son:

La probabilidad del cluster 1 es $2/3$ y la del cluster 2 es $1/3$ por lo tanto necesitamos 2.7549 bits para representar los tres símbolos: CL1, CL2, CL1. Para

Cluster	Node	Prob
1	A	3/14
1	B	4/14
1	C	5/14
1	EXIT	2/14
2	D	3/7
2	E	1/7
2	F	2/7
2	EXIT	1/7

Table 16.3: Ejemplo con dos clusters

el cluster 1 necesitamos 26.9384 bits y para el 2 12.8966 bits el total es: 42.5893 bits. Es decir que particionando la red en dos clusters necesitamos menos bits para representar nuestro random-walk que los que usábamos suponiendo que toda la red era un único cluster. Los bits que perdemos indicando que salimos de un cluster y entramos a otro los ganamos al poder representar con menor cantidad de bits a los nodos dentro de cada cluster.

Hemos comentado cómo podemos usar la ecuación de infomap para estimar si una cierta partición de la red en comunidades es mejor que otra: simplemente calculando cuántos bits necesitaríamos para codificar un random-walk suficientemente largo como para considerarlo significativo. Tenemos que ver ahora de que forma construir un algoritmo que use esto y la respuesta es que vamos a usar exactamente el mismo método que usamos en el algoritmo de Louvain con la diferencia que en Louvain usábamos la modularidad para comparar clusterizaciones y ahora usaremos la ecuación de Infomap.

Algorithm 39: Infomap

```

1 Inicializar ListOfclusters con cada nodo de la red
2 bestResult = InfoMap(ListOfClusters)
3 lastResult = Inf
4 While(bestResult < lastResult) lastResult = bestResult
5 for Node in ListOfClusters do
6   bestNode = []
7   for Node2 <> Node in ListOfClusters do
8     NewListOfclusters= merge(Node,Node2,ListofClusters)
9     if newResult=Infomap(NewListOfClusters) < bestResult then
10       bestNode = Node2
11       bestResult = newResult
12   ListOfclusters = merge(Node,bestNode,ListofClusters)
13 Return ListOfclusters

```

El algoritmo comienza suponiendo que cada nodo de la red es un cluster.

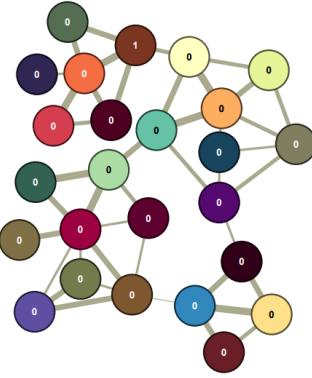


Figure 16.27: Estado inicial de Infomap, cada nodo es un cluster.

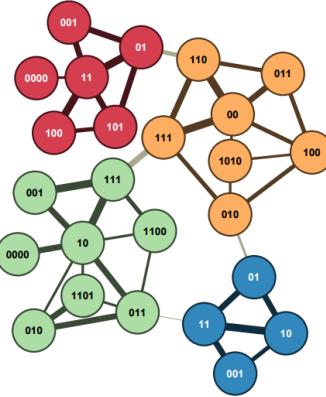


Figure 16.28: Estado final del algoritmo, la longitud de los random-walks se minimiza particionando la red en las 4 comunidades que mostramos.

Luego por cada nodo de la red hace un merge del mismo con el cluster que le da una mayor disminución en la longitud del random-walk, este proceso se repite hasta que no haya ninguna opción de merge posible.

La diferencia fundamental entre el algoritmo de Louvain (modularidad) e Infomap es que el primero se basa en la estructura topológica de la red mientras que el segundo se basa en el flujo dentro de la misma. En redes en las cuales las aristas expresan simplemente una semejanza entre nodos el algoritmo de Louvain suele dar mejores resultados mientras que en redes en las cuales el grafo expresa el flujo de información dentro de la misma Infomap suele tener resultados superiores. En una red social se pueden dar ambas cosas al mismo tiempo, pero en general las aristas expresan el flujo de información, los mensajes en Facebook o Twitter circulan a través de las aristas de la red, en estos casos es de esperar que Infomap produzca buenos resultados.

16.7.6 Comunidades con Solapamiento: Big Clam

Estudiaremos el caso en el cual un nodo puede pertenecer a mas de una comunidad, es decir que existe solapamiento entre las comunidades de la Red. Este caso es el mas realista en Redes Sociales ya que es frecuente que un usuario pertenezca, al mismo tiempo, a varias comunidades. Por ejemplo a sus familiares, amigos, compañeros de trabajo, compañeros de hobby, etc.

El algoritmo que estudiaremos llamado BigClam [Leskovec] es eficiente para encontrar comunidades con solapamiento en Redes grandes.

Vamos a necesitar el número de comunidades que queremos encontrar k

como parámetro. Cada comunidad tendrá asociada una probabilidad P_k de que un nodo pertenezca a ella. La probabilidad de que dos nodos estén unidos está dada entonces por la probabilidad de que exista un link en alguna de las comunidades que ambos integran.

$$P(u, v) = 1 - \prod_{c \in M_u \cap M_v} (1 - p_c) \quad (16.14)$$

La fórmula parte de la probabilidad de que un nodo no pertenezca a una comunidad $(1 - P_c)$, la probabilidad de que un nodo no pertenezca a ninguna comunidad es entonces la productoria de estas probabilidades. La probabilidad de que exista el link entre u y v es 1 menos la probabilidad de que u y v no estén juntos en ninguna comunidad.

Lo que queremos es encontrar el modelo que maximice la semejanza con el grafo de nuestra red social que conocemos:

$$P(G|\Omega) = \prod_{(u,v) \in F} P(u, v) \prod_{(u,v) \notin G} (1 - P(u, v)) \quad (16.15)$$

Es decir que queremos maximizar $P(u, v)$ cuando en el grafo existe una arista entre u y v y queremos minimizar $P(u, v)$ cuando la arista no existe realmente en el grafo.

Definimos F_{ua} a la fuerza de conexión del nodo u con la comunidad a . Por lo tanto para la comunidad a la probabilidad de que u y v se conecten es:

$$P_A(u, v) = 1 - \exp(-F_{ua}F_{va}) \quad (16.16)$$

Calculamos entonces $P(u, v)$

$$\begin{aligned} P(u, v) &= 1 - \prod_c (1 - P_c(u, v)) \\ &= 1 - \exp\left(-\sum_c F_{uc}F_{vc}\right) \quad (16.17) \\ &= 1 - \exp(-F_u F_v^t) \end{aligned}$$

Supongamos que tenemos 4 comunidades (a,b,c,d) y 3 nodos: u, v, w . Sea la siguiente matriz F :

$$\begin{pmatrix} a & b & c & d \\ 0 & 1.2 & 0 & 0.2 \\ 0.5 & 0 & 0 & 0.8 \\ 0 & 1.8 & 1 & 0 \end{pmatrix} \begin{matrix} u \\ v \\ w \end{matrix}$$

Por lo tanto $F_u F_v^t = (0, 1.2, 0, 0.2) * (0.5, 0, 0, 0.8)^t = 0.16$ y $P(u, v) = 1 - \exp(-0.16) = 0.14$ por otro lado $P(u, w) = 0.88$ y $P(v, w) = 0$

Podemos representar ahora el algoritmo como un problema de optimización (maximización):

$$l(F) = \sum_{(u,v) \in E} \log(1 - \exp(-F_u F_v^t)) - \sum_{(u,v) \notin E} F_u F_v^t \quad (16.18)$$

Esta función se puede optimizar por Gradient Descent:

$$\nabla l(F_u) = \sum_{v \in N(u)} F_v \frac{\exp(-F_u F_v^t)}{1 - \exp(-F_u F_v^t)} - \sum_{v \notin N(u)} F_v \quad (16.19)$$

Donde $N(u)$ indica los vecinos de u en el grafo original.

El algoritmo entonces realiza lo siguiente:

Algorithm 40: Big Clam

- 1 Inicializar F al azar
 - 2 **for** ($u \in G$) **do**
 - 3 $F_u = F_u - \alpha \nabla(F_u)$
 - 4 Si algun elemento de F es negativo lo convertimos en 0 (no hay probs negativas)
-

Luego de optimizar F tenemos que la matriz de probabilidades es:

$$1 - \exp(-FF^t) \quad (16.20)$$

En la matriz F tenemos la probabilidad de que cada nodo del grafo pertenezca a cada una de las k comunidades.

16.8 Análisis de comunidades en Redes Sociales

Intentemos ahora evaluar cada comunidad de una Red Social, ¿cómo podemos medir si una comunidad tiene mejor estructura que otra?. Para que un cluster sea bueno tiene que tener muchas aristas internas y muy pocas aristas que salgan del cluster. Esto se define como la "conductancia" de un cluster.

$$\phi(S) = \frac{|\{(i,j) \in E; i \in S, j \notin S\}|}{\sum_{s \in S} k_s} \quad (16.21)$$

Contamos la cantidad de links que salen del cluster y dividimos por la sumatoria de los grados de los nodos del cluster.

Para el cluster S tenemos 3 links que salen del cluster y la sumatoria de los grados es: 16 por lo que la conductancia es $3/16 = 0.1875$ en el caso de S' tenemos 2 aristas que salen del cluster y la sumatoria de grados es: 19 por lo que la conductancia es: 0.105. Cuanto menor sea la conductancia mejor será la calidad del cluster.

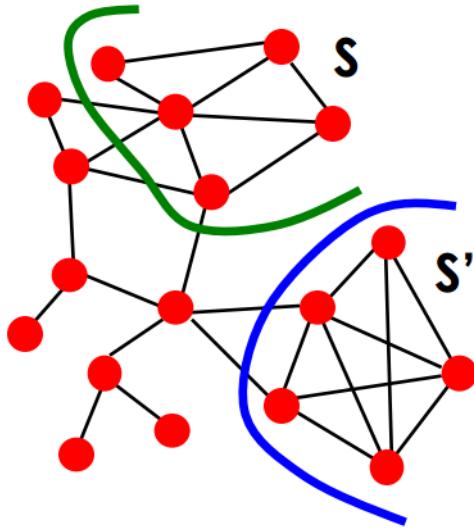


Figure 16.29: Ejemplo Conductancia

Para analizar la estructura de una Red podemos realizar un plot del mejor valor de conductancia para una comunidad de k nodos en función de k . Es decir que para cada valor de k tomamos el mínimo de todas las conductancias de clusters de tamaño k

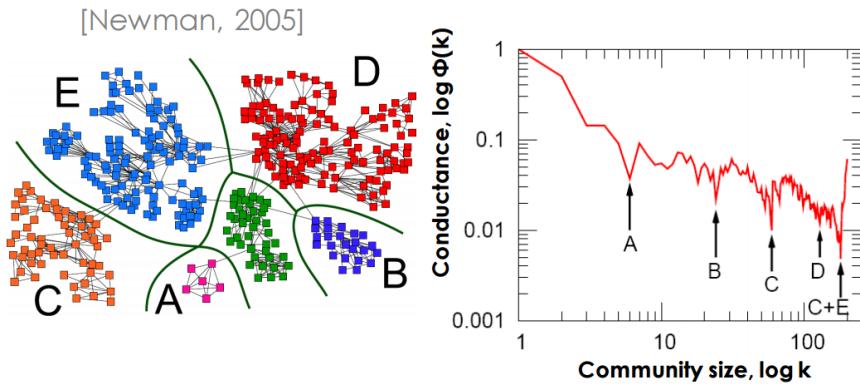


Figure 16.30: NCP plot

El plot NCP muestra picos de "calidad" para las verdaderas comunidades de la Red, es decir que podemos usarlo para encontrar cuáles son las verdaderas comunidades de nuestro grafo. Estos "valles" corresponden a las comunidades

que podemos ver visualmente en el grafo de la Red.

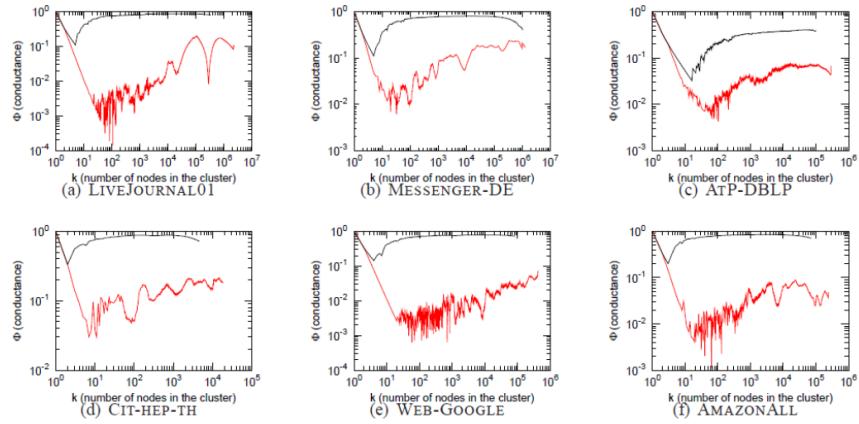


Figure 16.31: NCP plots

En todos los casos vemos que la calidad de los clusters mejora a medida que estos tienen mayor cantidad de miembros pero luego de un cierto umbral empeoran dramáticamente. La explicación es que a medida que el tamaño de los clusters aumenta la cantidad de aristas dentro de cada cluster crece de forma menor a la cantidad de aristas que salen de cada cluster, por esto la conductancia empeora.

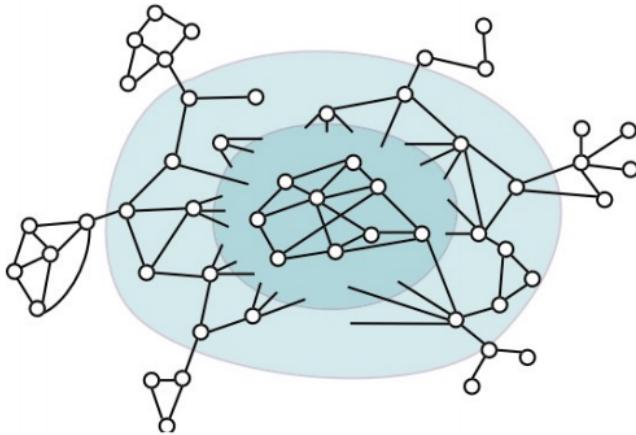


Figure 16.32: Estructura de una Red Social

Esto quiere decir que los mejores clusters apenas están conectados con el

resto de la red. Esto da origen a una explicación sobre la estructura de las Redes Sociales en donde podemos ver a la red como un núcleo y un conjunto de comunidades satelitales apenas unidas al núcleo. Estas comunidades satélites son las mejores comunidades de la Red.

16.9 Cascadas en Redes Sociales

Estudiaremos ahora el flujo de la información en las Redes Sociales, en concreto la forma en la cual la información se difunde a través de la topología de la red. Esto es importante en temas como marketing, prevención y atención de catástrofes, conversión electoral y muchas aplicaciones mas. La difusión de la información se inicia cuando un usuario de la red genera el primer mensaje sobre dicho tema, este mensaje se va propagando por la topología de la red y llega alcanza un cierto número de nodos (usuarios), al conjunto de nodos a los cuales llega la información lo llamaremos una *cascada*.

Para poder estudiar cascadas en redes sociales es necesario contar con un modelo de difusión, es decir un criterio que hace que un nodo comparta la información que le ha llegado. Vamos a usar un modelo de difusión muy simple que es "si al menos $q\%$ de mis vecinos difunden algo yo también lo hago" vamos a decir que esto nos define una cascada q .

Nos interesa estudiar cuándo se detiene una cascada en una red social y como maximizar la influencia, es decir como llegar a la mayor cantidad de nodos posibles.

16.9.1 Cascadas

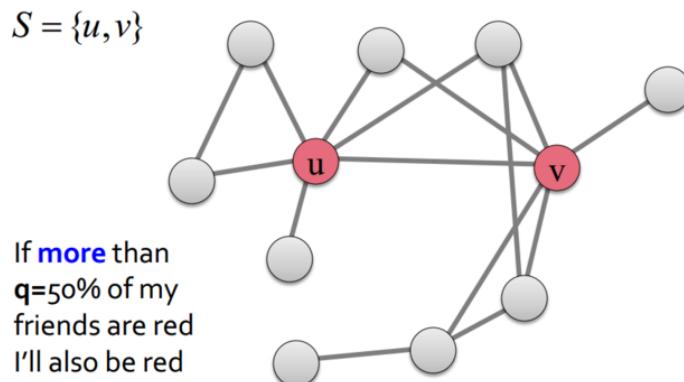


Figure 16.33: Cascadas en Redes Sociales

En nuestro primer ejemplo tenemos que los nodos u y v inician la cascada y

$q = 0.5$ con la condición de que mas del 50% de los vecinos tienen que difundir la información para que un nodo la difunda. rápidamente podemos ver que en el instante $t + 1$ los dos nodos que están por encima de u y v van a difundir la cascada ya que $2/2$ o $2/3$ de sus vecinos lo hacen.

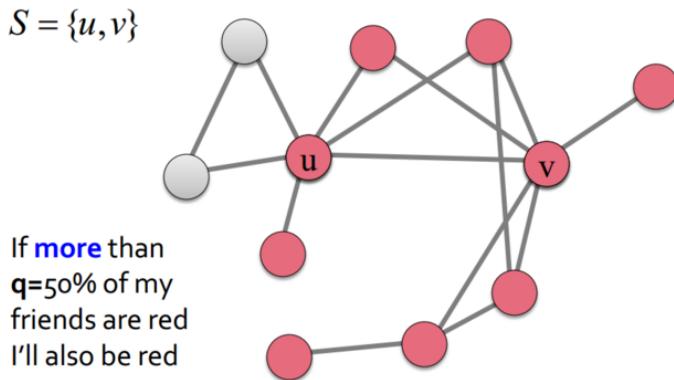


Figure 16.34: Cascadas en Redes Sociales

Continuando de esta forma llegaremos a que únicamente los dos nodos a la izquierda del grafo no difunden la información para estos nodos solo u difunde y nunca tendrán mas de un 50% de vecinos difundiendo el mensaje.

Nos interesa estudiar por qué se detiene una cascada. Para ello vamos a definir que un cluster tiene densidad p si todos los nodos del cluster tienen al menos p aristas dentro del mismo.

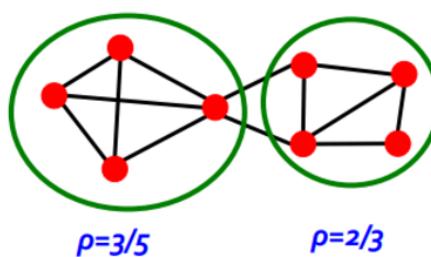


Figure 16.35: Densidad de un Cluster

En el ejemplo vemos que para el primer cluster todos los nodos tienen al menos 3 de sus aristas dentro del cluster. Por lo tanto la densidad es $3/5$. En

el segundo caso p es igual a $2/3$. Vemos que es el mínimo porcentaje de aristas que queda dentro del cluster para cualquier nodo.

El p mínimo de un cluster nos determina el q máximo. Es decir que para que una cascada q no se detenga no puede haber un cluster con densidad $p > (1-q)$. En nuestro caso el p mínimo entre todos los clusters es $3/5$ por lo tanto el q máximo es $2/5$, es decir que para que la información se difunda a todos los nodos esta tiene que compartirse a lo sumo si $2/5$ de los vecinos de un nodo lo hacen. Es decir que un q de $3/5$ o $4/5$ no alcanzaría todos los nodos del grafo.

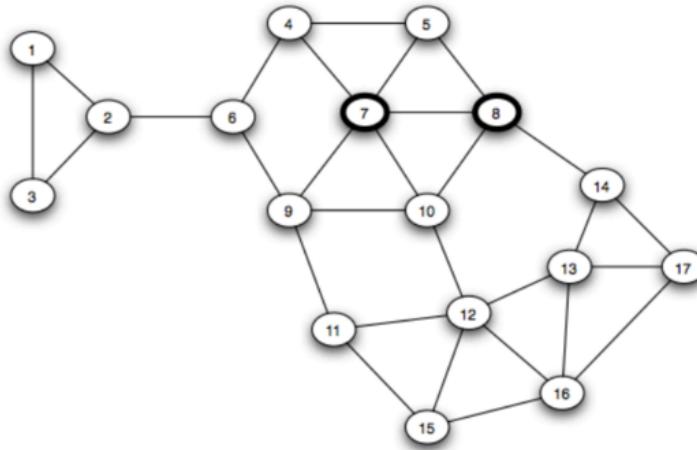


Figure 16.36: Difusión de Cascadas

En el ejemplo y con $q = 2/5$ la información se difunde a todos los nodos del cluster $[4,5,6,7,8,9,10]$ pero ahí se detiene, los otros nodos nunca llegan a tener $2/5$ de sus vecinos difundiendo la información.

La conclusión es que los clusters: las comunidades son obstáculos para la difusión de la información en las redes sociales.

16.9.2 Maximización de Influencia

Maximizar influencia implica llegar a la mayor cantidad de nodos posibles en una red mediante el uso de la menor cantidad de nodos posibles. Es decir que nos interesa descubrir un conjunto pequeño de nodos influyentes de forma tal que la información llegue a la mayor cantidad de usuarios posibles. Esto tiene, claramente, todo tipo de aplicaciones, desde campañas de marketing hasta campañas electorales o simplemente publicidad.

Lamentablemente el problema formal es NP-Completo, se trata del conocido problema de Set-Cover en donde queremos encontrar el conjunto mínimo de vértices de forma tal que estos cubran todas las aristas del grafo. Es necesario aproximar el problema.

Vamos a definir a f como la función que recibe un conjunto de nodos y nos devuelve la "influencia" de los mismos. Esta función es una función *submodular*, esto quiere decir que si agregamos un nodo a un conjunto el resultado nos da menor o igual ganancia que agregarlo a un subconjunto.

Supongamos que tenemos el conjunto de nodos $\{A, B, C, D\}$ y queremos agregar $\{E\}$, la ganancia de influencia es $f(\{A, B, C, D, E\}) - f(\{A, B, C, D\}) = f_1$. Por otro lado tomemos el conjunto $\{A, B, C\}$ y agreguemos el nodo $\{E\}$, la ganancia de influencia es ahora: $f(\{A, B, C, E\}) - f(\{A, B, C\}) = f_2$. Es evidente que f_2 tiene que ser al menos mayor o igual a f_1 . El motivo es muy simple: lo que el nodo E nos aporta en el segundo caso siempre va a ser mayor o igual a lo que nos aporta en el primero donde no teníamos a D , todo depende de que tanto se superpone la influencia de E y D , en ningún caso podemos ganar mas influencia agregando un nodo a un super-conjunto con respecto a cualquier conjunto original.

Sea $S \subseteq T$

$$f(S \cup \{u\}) - f(S) \geq f(T \cup \{u\}) - f(T) \quad (16.22)$$

El primer término es lo que ganamos agregando u a un conjunto chico y el segundo término la ganancia de agregar u al conjunto mayor.

Cuando f es una función monótona y sub-modular un algoritmo tipo greedy es en el peor caso un 63% de la solución óptima. Esto es bastante aceptable por lo que podemos usar un algoritmo greedy muy simple para maximizar influencia: en cada paso agregamos el nodo que mayor ganancia de influencia nos aporte.

El problema es que la función f que mide la influencia de un nodo en realidad no existe. Afortunadamente esta función se puede simular, mediante un modelo de cascadas y la topología de la red podemos estudiar a cuantos nodos llega la información a partir de un cierto conjunto de nodos, agregando siempre en cada paso el nodo que mayor ganancia de influencia nos da.

En definitiva se trata de resolver un problema NP-Completo mediante un algoritmo greedy que optimiza una función que en realidad no existe (!!).

16.10 Redes Sociales con Signo

En ciertas redes sociales las relaciones entre los usuarios pueden expresarse de forma positiva o negativa. Por ejemplo votos a favor o en contra de una propuesta del usuario, amigos y enemigos en una red social, confianza o desconfianza en un sitio de ventas online, etc. En esta sección analizaremos un algoritmo para predecir el signo de una relación que aun no existe entre dos usuarios, decir poder predecir si dos usuarios van a ser amigos o enemigos o si un usuario va a votar a favor o en contra de la propuesta de un determinado usuario.

Vamos a analizar las relaciones triangulares es decir entre tres usuarios de la red. Hay dos triángulos que son naturalmente estables (balanceados) que son los que obedecen a las siguientes reglas:



Figure 16.37: Relaciones Balanceadas

1. Los amigos de mis amigos son mis amigos
2. Los enemigos de mis enemigos son mis amigos



Figure 16.38: Relaciones No Balanceadas

Otras relaciones son inestables, por ejemplo si tengo dos amigos pero estos son enemigos entre sí. O un grupo de tres usuarios en los cuales todos son enemigos de todos. Explorando estos 4 triángulos en redes sociales podemos analizar cuantas veces aparece cada uno en relación a cuantas veces aparecería si los signos de las aristas fueran completamente aleatorios. El resultado es que las relaciones balanceadas son mas frecuentes que en el caso aleatorio y las relaciones desbalanceadas son menos frecuentes aunque en el caso de las tres enemistades en realidad la desviación es mínima.

El caso mas dramático es el triángulo no balanceado en donde A es amigo de B y C pero B y C son enemigos. Esto es lógico ya que va en contra del mecanismo de clausura triangular que hemos estudiado y es tan popular en las redes sociales, cuando un usuario tiene dos amigos la tendencia es que estos

Triángulo	Epinions	Wikipedia
balanced 1	+0.25	+0.21
balanced 2	+0.02	+0.11
unbalanced 1	-0.27	-0.41
unbalanced 2	-0.002	+0.0001

Table 16.4: Relaciones triangulares en Wikipedia y Epinions

formen una amistad, que formen una relación de enemigos es todavía menos probable a que no se relacionen en absoluto.

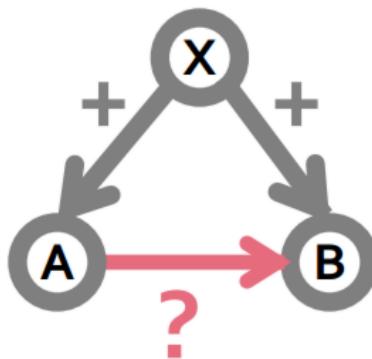


Figure 16.39: Predecir el Signo de una relación

Nos interesa predecir el signo de una relación faltante en un triángulo, es decir descubrir como se van a llevar dos usuarios en base a un usuario en común.

Existen 16 casos a analizar en donde nos falta el signo de una relación (marcada en color rojo). Lo que vamos a hacer es contar cuántos triángulos de cada tipo cerraría la relación faltante. Tenemos un set de entrenamiento en donde podemos ver para cada uno de los 16 contextos posibles cuál es la predicción para la relación faltante. Con este set de entrenamiento podemos usar Logistic Regression o un SVM lineal para predecir el signo de la relación A-B para cada uno de los triángulos que cerraría y luego simplemente definimos por mayoría.

En nuestro ejemplo la relación faltante A-B cierra varios triángulos, tenemos que correr para cada uno de estos triángulos una predicción del signo que falta y luego ver si ganan los positivos o los negativos.

Este algoritmo alcanza entre un 81% y hasta un 95% de precisión para predecir el signo de las relaciones faltantes entre los usuarios. Por ejemplo 94.4% de efectividad en Slashdot y 81% de efectividad en Wikipedia en donde aparentemente las relaciones entre los usuarios son mas complejas que en Slashdot.

Es interesante que la estructura de la red alcance sea lo único necesario para

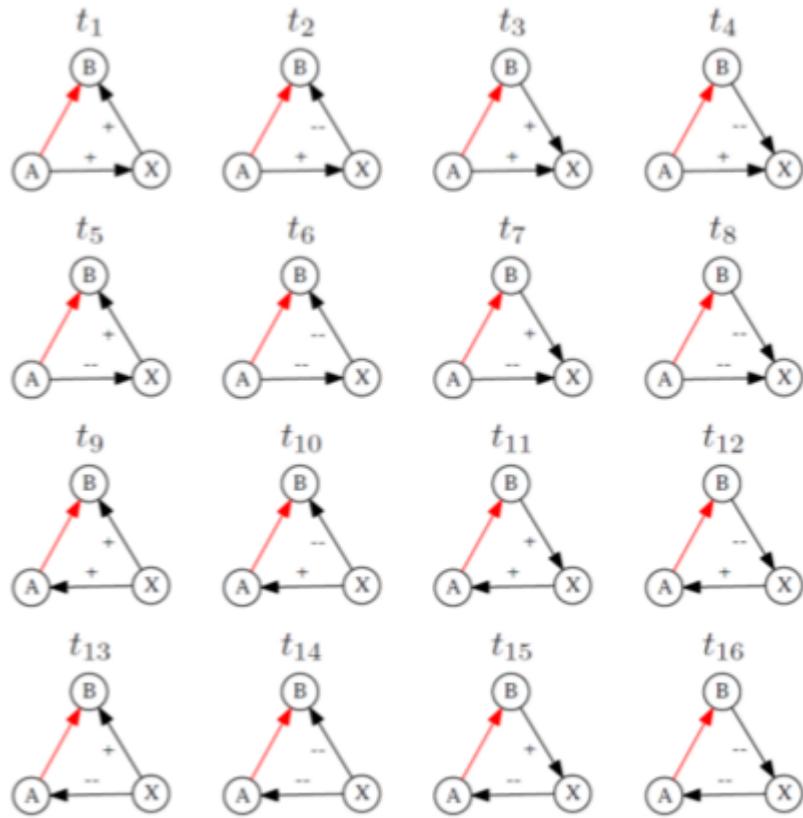


Figure 16.40: Contextos posibles

predicir la relación entre dos usuarios de la misma que aun no se conocen.

16.11 Motifs

Un "motif" es un sub-grafo de k nodos. Es decir alguno de todos los posibles grafos que podemos formar con k nodos, por ejemplo con 3 nodos y aristas dirigidas hay exactamente 13 sub-grafos. Para 4 nodos la cantidad es mas de 100.

La teoría es que ciertos motifs ocurren con mayor frecuencia en las Redes Sociales que en redes aleatorias. Para comprobar esto podemos construir un grafo aleatorio G_{np} de igual cantidad de nodos y aristas que nuestra red y contar la cantidad de motifs de cada tipo en nuestra red y en el grafo aleatorio. La diferencia sirve para caracterizar la red.

Esto sirve para comparar redes sociales entre sí y ver si la forma en la que

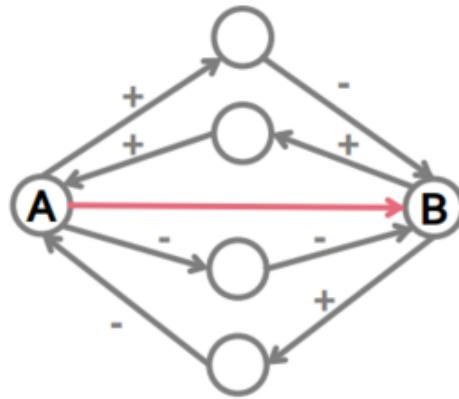
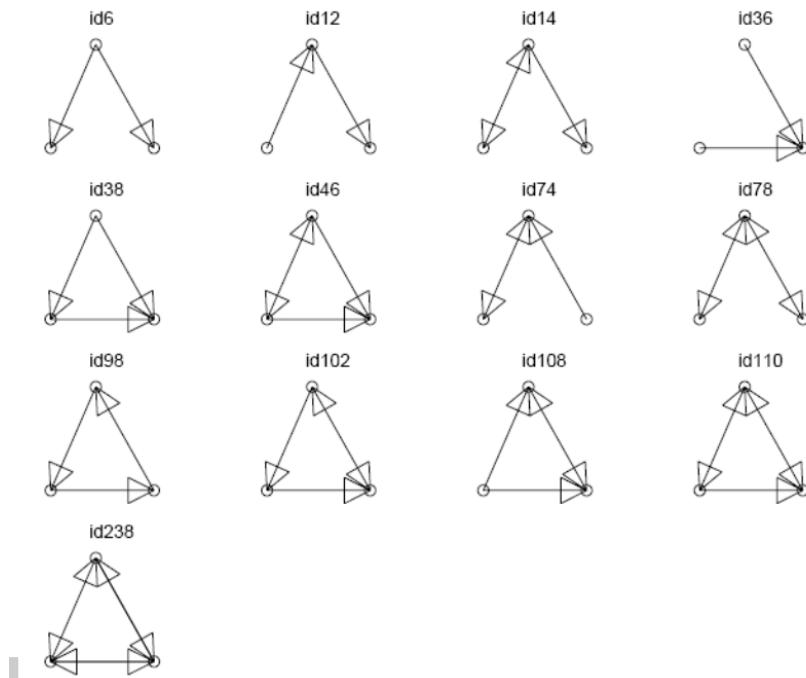


Figure 16.41: Triángulos para la relación A-B

Figure 16.42: Motifs con $k = 3$

se distribuyen los motifs es similar en una u otra red.

En el ejemplo vemos 19 redes y sus profiles de acuerdo a la aparición de

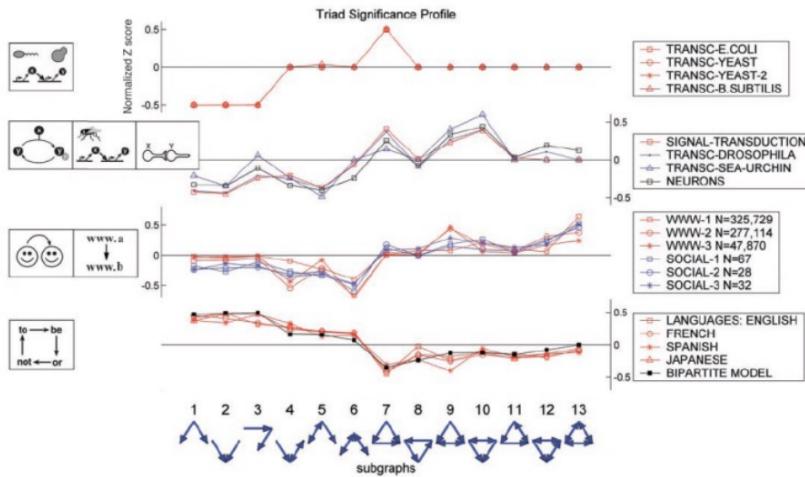


Figure 16.43: Motifs en Redes Sociales

los 13 motifs de 3 nodos en cada una de ellas. Podemos ver que la Web tiene características similares a las redes sociales y la principal característica es un profundo rechazo al motif número 6 en donde tenemos el caso de que A sigue a B y B sigue a A, A sigue a C y C sigue a A pero B no sigue a C ni C sigue a B. Esto es un desafío a la clausura triangular y sabemos que las redes sociales tienen una fuerte debilidad por la misma.

Existe software que permite obtener el profile de una red mediante sus motifs y esto permite caracterizar a la red y compararla contra otras redes independientemente de sus dimensiones, de esta forma podemos agrupar las redes en familias y esto permite saber que los algoritmos que funcionan bien en una determinada red probablemente funcionen bien en otras redes de la misma familia pero no en familias diferentes.

16.12 El Lenguaje en las Redes Sociales

En una red social podemos estudiar no solo la topología de la misma sino el contenido que los usuarios escriben. Estos contenidos pueden ser posts, mensajes, tweets, reviews, comentarios, preguntas, respuestas, etc. Lo que queremos estudiar es el lenguaje utilizado en estos posts. Por ejemplo podemos analizar el porcentaje o probabilidad de aparición de una palabra en cualquier mensaje de la red a lo largo del tiempo. Observaremos que ciertas palabras son "adoptadas" por la comunidad y se vuelven más populares en detrimento de otras palabras que pasan de moda y dejan de usarse. En definitiva los usuarios tienden a copiar el lenguaje de otros usuarios, esto ocurre de forma involuntaria.

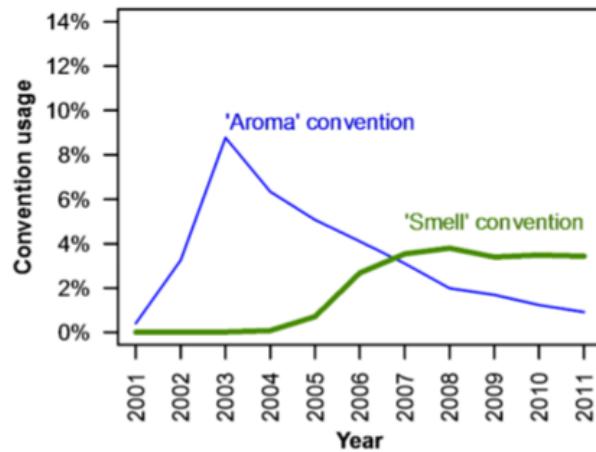


Figure 16.44: El lenguaje en las redes sociales

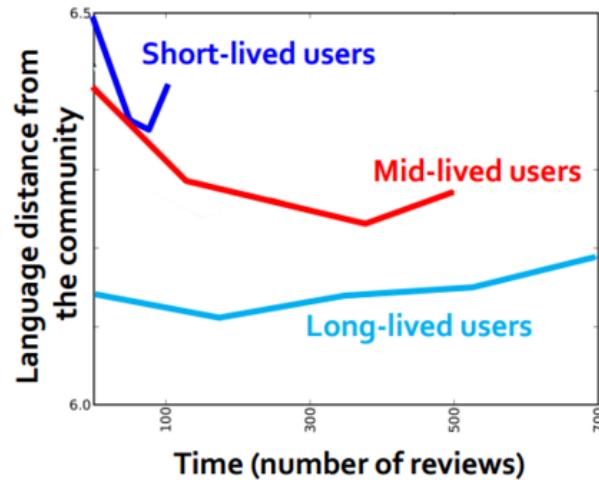


Figure 16.45: Evolución del lenguaje en Redes Sociales

Podemos hacer un promedio de todos los mensajes de la red y esto nos dará el modelo de lenguaje promedio de la red social. Podemos entonces comparar el lenguaje de un determinado usuario contra el lenguaje de la red a lo largo del tiempo. Lo que observamos es que los usuarios se van adaptando al lenguaje de

la red y la "distancia" entre el lenguaje del usuario y el lenguaje promedio de la red va disminuyendo a lo largo del tiempo.

Esto ocurre hasta que en un determinado instante el usuario empieza a separarse del lenguaje promedio de la red, el motivo por el cual esto pasa es porque el usuario queda atrapado en el pasado, su lenguaje evoluciona hasta un punto tal en que ya no puede modificarlo y a partir de ahí sus posts van a estar cada vez mas lejos del lenguaje promedio de la red.

Algo muy interesante es que esto es independiente de cuanto tiempo vaya a permanecer el usuario en la red. En el momento en que el lenguaje del usuario comienza a distanciarse del promedio sus días están contados. Esta tendencia se verifica en foros, sitios de reviews y sitios con preguntas y respuestas, eventualmente también en grupos de Facebook considerando a cada grupo como una red social independiente.

Esto permite varias cosas. En primer lugar podemos predecir la longevidad del usuario en la red, una curva de lenguaje que sigue acercándose al promedio de la red es señal de que el usuario es un participante activo de la red y que todavía tiene capacidad de aprender de la misma. Un usuario que rápidamente se distancia del lenguaje promedio es un usuario que en general dura muy poco en la red. Este mecanismo sirve, por ejemplo, como método de detección de trolls, ya que estos en ningún momento se preocupan por adaptarse al lenguaje promedio de la red.

16.13 Evolución de una Red Social

Analizaremos ahora la evolución de la topología de una red social a lo largo del tiempo. Nos interesa analizar de que forma evolucionan las diferentes métricas que estudiamos al principio del capítulo a medida que la red social crece.

16.13.1 Cantidad de Nodos y Aristas

Queremos ver de que forma evoluciona la cantidad de aristas de una red en función de la cantidad de nodos. Una suposición es decir que el grado promedio se va a mantener constante a lo largo del tiempo. Esto no funciona así en las redes sociales.

A medida que la cantidad de nodos aumenta la cantidad de aristas aumenta aun mas, es decir que el grado promedio de la red aumenta a lo largo del tiempo. Este proceso se denomina "densificación" y responde, adivina adivinador, a una ley de potencias.

La densificación se puede expresar mediante una ley de potencias de la forma:

$$E(t) \approx N(t)^\alpha \quad (16.23)$$

o lo que es igual

$$\frac{\log(E(t))}{\log(N(t))} = \text{const} \quad (16.24)$$

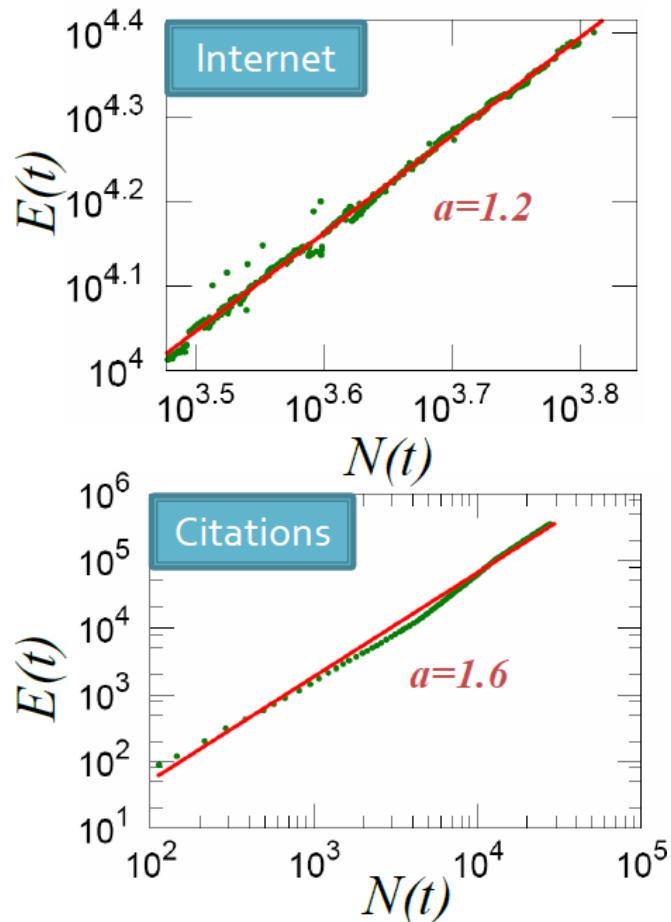


Figure 16.46: Cantidad de aristas en función de la cantidad de nodos, escala logarítmica

En donde α es el exponente de densificación. Para la web es $\alpha = 1.2$ y para la red de citaciones en papers tenemos $\alpha = 1.6$ observemos que en escala logarítmica tenemos una recta casi perfecta lo cual indica que la ley de potencias se cumple casi de forma perfecta.

En general el exponente de densificación es mayor a 1 e inferior a 2. Cuando $\alpha = 1$ tenemos una densificación lineal (las redes sociales no funcionan así) y cuando $\alpha = 2$ la densificación es cuadrática y el resultado es un grafo fuertemente conexo, las redes sociales tampoco son fuertemente conexas. Por eso el exponente siempre es mayor a 1 y menor a 2.

16.13.2 Diámetro

La intuición es que a medida que se agregan nodos en una red el diámetro de la misma crece lentamente por ejemplo en el orden de $O(\log N)$. Esto no ocurre en redes sociales sino todo lo contrario. El diámetro disminuye a lo largo del tiempo.

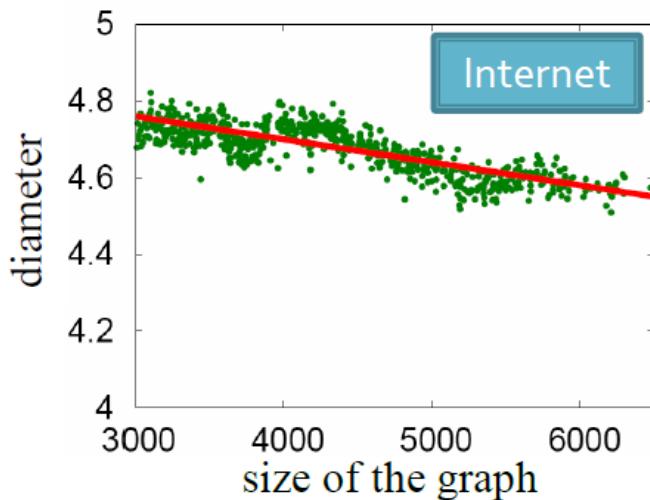


Figure 16.47: Evolución del diámetro en una red social

Esto quiere decir que a medida que la red crece la distancia entre los nodos lentamente disminuye. En primer lugar deberíamos analizar si esto es consecuencia de la densificación. Haciendo el experimento en una red aleatoria usando un exponente de densificación $\alpha = 1.3$ observamos que el diámetro en realidad crece a lo largo del tiempo por lo tanto tiene que haber otra explicación.

La explicación está dada por la combinación de la densificación y la forma en la que se distribuye el grado en una red social, si comparamos la evolución del diámetro en una red social y una red aleatoria con igual exponente de densificación e igual distribución del grado entonces el diámetro disminuye exactamente de la misma forma en ambas.

En conclusión la forma en que se forman los links (preferential attachment) y la densificación son los factores que hacen que el diámetro de las redes sociales disminuya a lo largo del tiempo.

La densificación es una ley de potencias con exponente menor a 2, por lo tanto su promedio es infinito, esto quiere decir que manteniendo constante el exponente de densificación α el grado promedio crece.

16.14 Grafos de Kronecker

Hemos visto varios modelos de redes que nos permiten reproducir algunas de las características de una red social. En esta sección vamos a intentar desarrollar un modelo generativo que nos permita hacer "crecer" un grafo de forma tal que el mismo tenga características similares a una red social. El modelo que vamos a usar está basado en grafos de Kronecker y consiste en replicar una pequeña estructura recursivamente como si fuera un fractal. La hipótesis es que cada parte de la red se parece a la red misma.

Empecemos definiendo una operación fundamental que vamos a usar que el el *producto de Kronecker* entre dos matrices A y B :

$$A_{n \times m} \otimes B_{k \times l} = \begin{pmatrix} a_{11}B & a_{12}B & \dots & a_{1m}B \\ a_{21}B & a_{22}B & \dots & a_{2m}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}B & a_{n2}B & \dots & a_{nm}B \end{pmatrix} \quad (16.25)$$

El resultado es una matriz de $nkxm$.

El producto de Kronecker entre dos grafos es el producto de Kronecker entre sus matrices de adyacencias.

Ejemplo:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Observemos que al multiplicar una matriz de Kronecker por si misma lo que obtenemos es una replicación de la matriz original de acuerdo al patrón de 1s y 0s en la misma. Recomiendo tomar un tiempito para estudiar este resultado y hacer algunos ejemplos mas para poder comprender el funcionamiento de la operación.

Tenemos entonces un proceso recursivo para expandir un grafo, nuestro grafo va a estar formado por:

$$K_m = K_1 \otimes K_1 \otimes K_1 \otimes \dots \otimes K_1 = K_{m-1} \otimes K_1 \quad (16.26)$$

Por ejemplo a partir de un cierto grafo generador y su matriz de adyacencias podemos obtener la matriz/grafos que vemos en color azul.

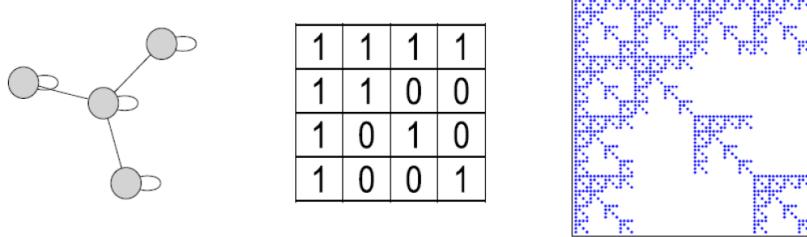


Figure 16.48: Grafos de Kronecker

16.14.1 Propiedades

La primera observación importante sobre los grafos de Kronecker es que si tenemos un generador K_1 con N_1 nodos y E_1 aristas entonces el grafo de Kronecker K_m tiene: N_1^m nodos y E_1^m aristas. Es decir que tenemos que la densificación es una ley de potencias (!). El exponente de densificación se puede calcular de la forma:

$$\alpha = \frac{\log(E(t))}{\log(N(t))} = \frac{\log(E_1^t)}{\log(N_1^t)} = \frac{\log(E_1)}{\log(N_1)} \quad (16.27)$$

Como sabemos que $E(t) > N(t)$ entonces $\alpha > 1$

De la misma forma se pueden probar las siguientes propiedades:

- La distribución del grado es una ley de potencias
- El diámetro es constante o disminuye
- La distribución de los autovalores es una ley de potencias

Estas propiedades no las demostraremos por razones de espacio (?) pero nos sirven para entender que los grafos de Kronecker tienen propiedades muy similares al grafo de una red social.

16.14.2 Grafos de Kronecker Estocásticos

En un grafo de Kronecker estocástico la matriz generadora no tiene 1s y 0s sino la probabilidad de cada arista. Llamaremos a la matriz generadora estocástica Θ_1 . Al igual que antes calcularemos Θ_k y luego cada celda de Θ nos dará la probabilidad de que nuestro grafo tenga cada una de las aristas posibles, en base a estas probabilidades construimos luego nuestro grafo.

Ejemplo:

$$\Theta_1 = \begin{pmatrix} 0.5 & 0.2 \\ 0.1 & 0.3 \end{pmatrix} \quad (16.28)$$

$$\Theta_2 = \Theta_1 \otimes \Theta_1 = \begin{pmatrix} 0.25 & 0.10 & 0.10 & 0.04 \\ 0.05 & 0.15 & 0.02 & 0.06 \\ 0.05 & 0.02 & 0.15 & 0.06 \\ 0.01 & 0.03 & 0.03 & 0.09 \end{pmatrix} \quad (16.29)$$

A partir de esta matriz podemos generar un grafo que cumpla con estas probabilidades por ejemplo:

$$G = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \quad (16.30)$$

Los grafos de Kronecker estocásticos a partir de una matriz generadora de tipo $\Theta_1 = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ tienen varias propiedades importantes:

- el grafo es conexo si $b + c > 1$
- El grafo tiene un componente conexo de tamaño $\Theta(n)$ si: $(a+b)(b+c) > 1$
- El diámetro es constante si: $b + c > 1$

16.14.3 Estimación de la matriz generadora

Queremos ahora resolver el problema de dado el grafo de una red social calcular cuál es la matriz generadora de forma tal que el grafo de Kronecker sea lo más parecido posible al grafo real. Es decir queremos maximizar $P(G|\Theta)$ usando, por ejemplo, gradient descent.

Podemos calcular $P(G|\Theta)$ de la forma:

$$P(G|\Theta) = \prod_{(u,v) \in G} \Theta_k[u, v] \prod_{(u,v) \notin G} (1 - \Theta_k[u, v]) \quad (16.31)$$

Es decir que para las aristas que realmente pertenecen al grafo queremos que su probabilidad en Θ_k sea cercana a 1 y para las aristas que no están en el grafo queremos que su probabilidad sea cercana a cero.

Un serio problema es que no sabemos cuál es la correspondencia entre los nodos en la matriz de Kronecker y la matriz de adyacencias real del grafo, y hay un total de $O(n!)$ correspondencias posibles. Esto no es nada bueno.

Para resolver este problema vamos a aplicar una variante del algoritmo Hill-Climbing que vimos en el capítulo 1, hace ya tanto tiempo.

Empezamos con una permutación al azar de los N nodos que define una correspondencia, por ejemplo [1,3,4,2] define que el nodo 1 del grafo es el primer nodo en la matriz de Kronecker, el nodo 2 del grafo es el cuarto nodo en la matriz de Kronecker, etc. A partir de esta permutación tomamos dos nodos al azar y los intercambiamos, calculamos $P(G|\Theta)$ si el resultado mejora entonces mantenemos el cambio y sino lo deshacemos. Esto lo repetimos hasta que no mejore la semejanza. Podemos repetir el proceso algunas veces, dependiendo del tiempo que querramos dedicarle, para evitar máximos locales.

16.14.4 Generando Redes Sociales con Grafos de Kronecker

El experimento que queremos hacer es dado el grafo de una red social G estimar la matriz generadora de Kronecker Θ_1 y luego en base a esta matriz generar un grafo de Kronecker artificial K . Finalmente queremos comparar las propiedades de G y K .

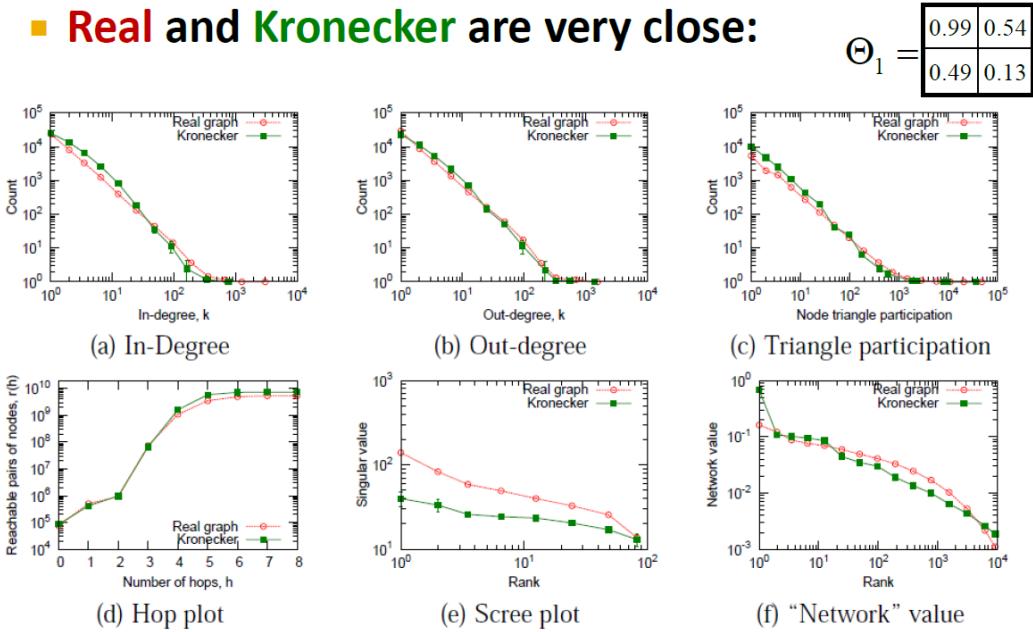


Figure 16.49: Grafos de Kronecker para Epinions

Por ejemplo para la red social de Epinions la matriz generadora es $\Theta_1 = \begin{pmatrix} 0.99 & 0.54 \\ 0.49 & 0.13 \end{pmatrix}$ y vemos que la mayoría de las propiedades del grafo real son prácticamente iguales en el grafo de Kronecker(!!). Por lo tanto hemos mostrado que una red social puede caracterizarse como la repetición recursiva de una

estructura muy simple, queremos ver que podemos aprender a partir de esta estructura:

$$\Theta_1 = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad (16.32)$$

■ **Small and large networks are very different:**

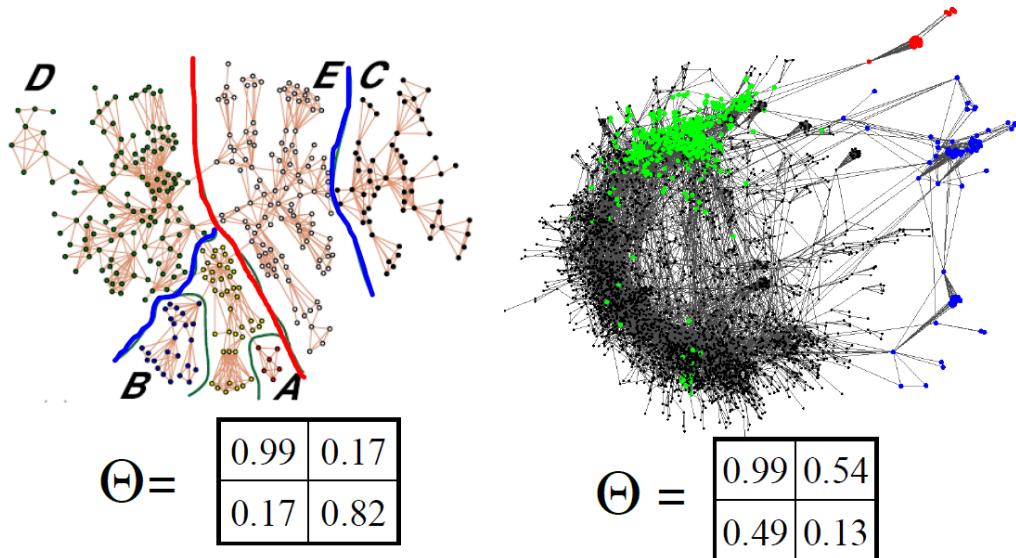


Figure 16.50: Grafos de Kronecker

Podemos ver que diferentes redes nos van a dar diferentes matrices generadoras. La estructura de una red la podemos caracterizar como una cebolla en donde cada capa de la misma se descompone en un núcleo y una periferia, recordemos que esta periferia es donde están las comunidades con mejor conductancia. Los valores de a, b, c y d nos permiten estudiar de que forma se va armando la estructura de la red y relacionarlos con la forma del núcleo y la periferia en la estructura real.

Chapter 17

Sistemas de Recomendaciones

Now you're looking for the secret,
but you won't find it, because of
course you're not really looking.
You don't really want to know.
You want to be fooled.

- The Prestige

17.1 Introducción: El modelo de descubrimiento

Un sistema de recomendaciones tiene como objetivo descubrir contenidos que puedan ser interesantes para el usuario. Estos contenidos pueden ser de cualquier tipo: películas, música, libros, noticias, productos, etc. En el apunte usaremos películas como contenido por conveniencia pero todos los temas que veremos se aplican de la misma forma a cualquier otro tipo de contenidos.

Los sistemas de recomendaciones en general tienen el enorme objetivo de transformar la forma en la que los usuarios utilizan internet cambiando un modelo basado en “búsquedas” por un modelo basado en “descubrimiento”.

En un modelo basado en búsquedas los usuarios usan motores de búsqueda como Google o Bing(?) para encontrar contenidos de interés, en un modelo basado en descubrimiento un sistema de recomendaciones descubre contenidos que pueden ser de interés para el usuario y se los acerca. Es decir que en lugar de hacer que el usuario busque los contenidos hacemos que los contenidos lleguen al usuario. El cambio filosófico es fundamental.

Una diferencia fundamental entre un modelo de búsqueda y un modelo de descubrimiento es que en una búsqueda los usuarios deben saber que es lo que quieren encontrar, en cambio en un modelo de descubrimiento el usuario encuentra cosas que no sabía que existían. Este tema es muy importante ya que

sin un sistema de recomendaciones adecuado los usuarios no podrían acceder a los contenidos que no conocen y como veremos más adelante en general hay muchos más contenidos desconocidos que conocidos.

17.2 Características de un Sistema de Recomendaciones

Un sistema de recomendaciones exitoso debe cumplir con algunas características mínimas que listamos a continuación:

- Precisión
- Serendipity
- Diversidad

17.2.1 Precisión

La precisión apunta a recomendarle al usuario contenidos que sea de su interés. Un buen sistema de recomendaciones no debe recomendar al usuario cosas que al mismo no van a interesarle, el efecto de ésto puede ser catastrófico ya que el usuario deja de confiar en el sistema e ignora los productos recomendados.

La mayoría de los sistemas de recomendación se enfocan fuertemente en lograr una alta precisión, ésto es muy importante pero, como veremos, no es lo único que debemos considerar.

17.2.2 Serendipity

El concepto de “serendipity” puede resumirse brevemente cómo: “no mostrarle al usuario contenidos que el usuario ya conoce o que puede descubrir fácilmente por sus propios medios”.

Esto es equivalente a decir que el sistema debe ser capaz de recomendarle al usuario contenidos que no son muy populares en general. Este principio, que es de gran importancia, está asociado a la forma en la cuál se distribuye la popularidad de los contenidos en todos los sistemas de recomendaciones. Se verifica como ley que existen unos pocos contenidos que son muy populares y una enorme cantidad de contenidos que tienen baja popularidad. Esto se conoce como el principio del “Long Tail”.

Si graficamos la cantidad de ítems en el eje “x” y la popularidad de los mismos en el eje “y” independientemente del tipo contenido obtenemos algo de la siguiente forma:

La distribución tiene la forma de un “power law” es decir $y = x^{\alpha}$ que no debemos confundir con una distribución exponencial.

El “long tail” hace referencia a la enorme cantidad de productos que no son muy populares y es a estos productos a donde debe apuntar un sistema de recomendaciones. Los motivos son varios. En primer lugar los productos

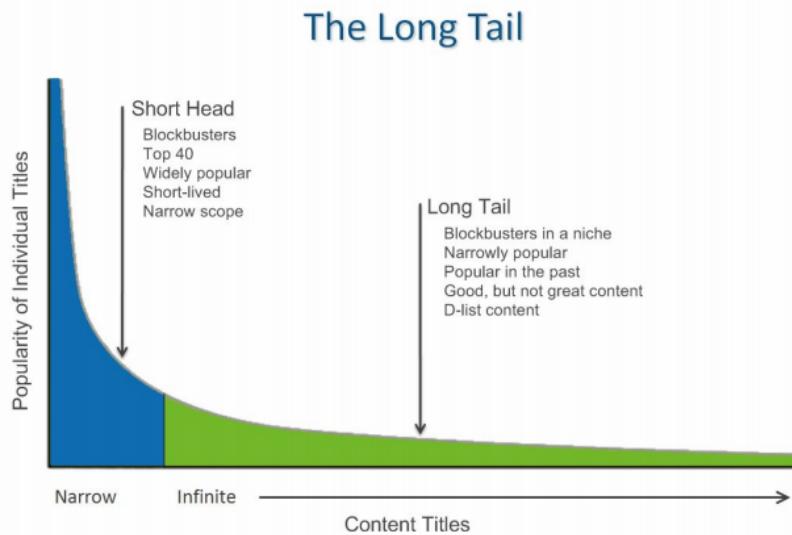


Figure 17.1: The Long Tail

populares son conocidos y el usuario o bien ya los conoce o bien puede encontrarlos fácilmente. En segundo lugar la cantidad de contenidos poco populares es muy grande por lo que recomendándolos de forma eficiente podemos incrementar mucho la utilidad de nuestro sistema. Finalmente hay que destacar que estos productos poco populares son el diferencial entre las ventas online y las ventas tradicionales, ya que los negocios tradicionales no suelen tener la enorme variedad que tienen los sistemas online



Figure 17.2: Ejemplo Long Tail

En este ejemplo podemos ver que la música de “Britney Spears” es muy

popular (el mundo está perdido), a los usuarios a quienes les gusta Britney suele gustarles Pink, pero recomendar música de Pink no tiene un gran atractivo porque Pink es una cantante muy popular y casi todos los usuarios la conocen. En cambio es más interesante recomendar temas de “No Doubt” o aun mejor de “The Selecter”. Si en éste preciso instante están buscando algún tema de “The Selecter” para ver de que se trata entonces habrán comprendido completamente el principio de “the long tail” y la importancia del principio de “serendipity” en un sistema de recomendaciones.

17.2.3 Diversidad

La diversidad apunta a no mostrarle al usuario contenidos que son exclusivamente de un mismo tipo. Si nuestros contenidos pudieran dividirse, de alguna forma, en categorías no es una buena idea recomendarle al usuario productos de una única y exclusiva categoría. Por ejemplo no sería bueno recomendarle al usuario únicamente películas bélicas.



Figure 17.3: Diversidad

17.2.4 Otros desafíos para un sistema de recomendaciones

A continuación listamos algunos problemas típicos en sistemas de recomendaciones. Algunos de éstos problemas son muy difíciles de solucionar y nos van a permitir entender la enorme complejidad que podemos enfrentar.

Los gustos de los usuarios pueden cambiar

Consideremos el caso de un usuario que venía comprando casi exclusivamente libros de ciencia ficción pero sus últimas tres compras son libros sobre historia del arte. ¿Qué conclusión podemos sacar sobre ésto?. Tal vez el usuario se ha hartado de la ciencia ficción y quiere ahora leer sobre arte en cuyo caso deberíamos recomendarle libros sobre arte. O tal vez el usuario es simplemente un fan de la ciencia ficción que tuvo que leer algunos libros sobre historia del arte para una materia que está estudiando en la facultad. En cuyo caso deberíamos

17.2. CARACTERÍSTICAS DE UN SISTEMA DE RECOMENDACIONES607

seguir recomendando libros de ciencia ficción. Como vemos éste problema es de difícil resolución porque en escenarios idénticos la mejor acción puede ser totalmente opuesta para dos usuarios.

Que el sistema de recomendaciones sea diverso en cierta forma nos ayuda a mitigar éste problema.

La influencia del tiempo

Los factores temporales son absolutamente críticos en los sistemas de recomendación. En primer lugar a medida que pasa el tiempo las calificaciones que los usuarios hicieron de los contenidos van envejeciendo. Si un usuario pensaba que una película era muy mala hace 2 años puede o no que siga pensando lo mismo pero sin dudas el peso de esta calificación no es tan relevante hoy en día como lo fue hace dos años.

Por otro lado se puede verificar que cuando una película es nueva sus calificaciones tienden a ser al principio más altas. Esto es fácil de explicar porque los usuarios consideran a que la película es una novedad como parte de la calificación. Una película que es muy esperada por los usuarios y que es muy mala tendrá al principio calificaciones mucho más altas que luego de un tiempo simplemente por el factor de ansiedad. Los usuarios tardan un cierto tiempo en darse cuenta que aquello por lo que tanto esperaron en realidad es muy malo.

El usuario a veces quiere ver cosas que no le gustan

Este es un problema particularmente molesto. Un usuario que califica positivamente películas de acción y bélicas a veces tiene ganas de ver una comedia romántica un domingo a la noche. El famoso caso de “voy a ver una película mala para reírme un rato”. Modelar este comportamiento en un sistema de recomendaciones es algo que, como podrán imaginar, resulta terriblemente difícil.

Lo que el usuario califica vs lo que el usuario ve

En los sitios en los cuáles el usuario puede calificar los contenidos se da a veces, en algunos usuarios (por suerte no en todos) un comportamiento muy particular. El usuario consume contenido de un determinado tipo y sistemáticamente lo califica con bajos puntajes. Por ejemplo el usuario ve en su mayoría comedias románticas pero les suele poner una o dos estrellitas. ¿Por qué pasa esto?

Lo que ocurre es que el usuario piensa que el sistema de recomendaciones es para realizar una crítica artística de las películas y no para decir su opinión personal sobre las mismas. “A mi esta película me gusta, pero como creo que no es muy buena le pongo 2 estrellitas”.

Esto puede modelarse comparando lo que el usuario consume con las calificaciones del mismo y establecer de esta forma un grado de “confianza” en las calificaciones del usuario que pueden usarse luego en el sistema de recomendaciones. A continuación veremos el primero y más importante de los algoritmos que suelen usarse en sistemas de recomendaciones.

17.3 Sistemas Basados en Contenidos (Content Based)

En un sistema de recomendaciones basado en contenidos el objetivo es recomendarle al usuario ítems similares a aquellos que le han gustado. Por cada ítem es necesario construir un *item profile* que es un vector que representa las características del ítem. Imaginemos que este vector es binario es decir que cada feature puede ser 1 o 0. Podemos usar como features cosas del estilo "película es de acción", "dirigida por Q. Tarantino", "duración menor a 120 minutos", "año de estreno 2014", etc. El vector puede tener realmente muchas dimensiones a veces miles.

Por cada usuario podemos también construir un *profile* que es un vector de igual cantidad de dimensiones, luego cuando un usuario califica una película sumamos el puntaje a aquellas dimensiones para las cuales la película tiene un "1" en su feature. Por ejemplo si el usuario califica con un "5" a "Pulp Fiction" vamos a sumar 5 a la dimensión que corresponde a "película de acción", "dirigida por Q. Tarantino", "con Bruce Willis", etc.

Teniendo el *profile* de cada ítem y de cada usuario podemos estimar la calificación de un usuario para un ítem calculando el coseno entre los *profiles*.

$$\cos\theta = \frac{xy}{\|x\|\|y\|} \quad (17.1)$$

Eventualmente hay que hacer algunas cuentas, por ejemplo podemos normalizar a todos los vectores para que tengan norma = 1 y entonces el resultado del coseno es una semejanza entre 0 y 1, si la semejanza va de 0 a 1 podemos multiplicar por la calificación máxima para estimar una calificación.

Los sistemas basados en contenido tienen varias ventajas:

- Solo necesitan la información del usuario para realizarle recomendaciones, es decir puede haber un único usuario y el sistema funciona.
- Tienen la habilidad de detectar los gustos particulares de los usuarios.
- Pueden recomendar ítems que sean nuevos o que no sean populares, solo se necesita tener el *profile* del ítem.
- Pueden explicar el motivo de sus recomendaciones, listando cuales son los features que hicieron que el ítem sea recomendado

Entre las desventajas podemos encontrar:

- Encontrar los features necesarios para construir los *profiles* es algo realmente difícil
- No puede recomendar nada a los usuarios nuevos ya que estos no tienen *profile*
- No pueden recomendar ítems que no estén dentro de los gustos del usuario, son sistemas que pueden quedar sobre-especializados

- No tienen la posibilidad de usar la opinión de otros usuarios sobre los ítems

Los sistemas basados en contenidos pueden funcionar muy bien pero son bastante difíciles de crear y mantener principalmente por la necesidad de seleccionar todos los features necesarios para construir un profile. En general no son tan populares como los sistemas basados en collaborative filtering.

17.4 Collaborative Filtering

Sean “n” usuarios y “m” ítems. Los usuarios califican los ítems con un número de 1 a 5. Podemos representar esto en una matriz de $n \times m$ que denominaremos “matriz de utilidad”



	SHERLOCK	HOUSE OF CARDS	THE AVENGERS	ARRESTED DEVELOPMENT	BREAKING BAD	THE WALKING DEAD
User 1	2		2	4	5	
User 2	5		4			1
User 3			5		2	
User 4		1		5		4
User 5			4			2
User 6	4	5		1		

Figure 17.4: Matriz de Utilidad

Notemos que la matriz tiene muchos “huecos” porque de todas las calificaciones posibles solo existen algunas pocas. Esto es en cierta forma similar a decir que la matriz es dispersa pero no es exactamente lo mismo ya que los elementos que faltan no son ceros sino que son números desconocidos.

El objetivo de Collaborative Filtering es estimar las calificaciones que nos faltan en la matriz de utilidad.

CF tiene dos formas de estimar las calificaciones que faltan:

- Semejanza entre usuarios
- Semejanza entre ítems

17.4.1 Semejanza User User

Supongamos que queremos estimar las calificaciones que le faltan al usuario “i”. Lo que hacemos es buscar los usuarios mas similares a “i” y luego estimar las calificaciones que faltan en base a un promedio ponderado de las calificaciones de los demás usuarios ponderadas de acuerdo a la semejanza que tengan con nuestro usuario “i”.

Necesitamos entonces una función de semejanza entre usuarios $\text{sim}(u,v)$ que calcule cuán similares son dos usuarios en base a las calificaciones que han realizado los mismos. Cuanto más parecidas sean estas calificaciones más parecidos serán los usuarios. Usuarios que han calificado conjuntos de ítems disjuntos tendrán semejanza nula.

Una de las funciones más usadas para calcular la semejanza es la función de correlación de Pearson:

$$\text{sim}(x, y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)(r_{ys} - \bar{r}_y)}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)^2} \sqrt{\sum_{s \in S_{xy}} (r_{ys} - \bar{r}_y)^2}} \quad (17.2)$$

Donde \bar{r}_x es el promedio de calificaciones del usuario x .

Notemos que si los promedios de calificaciones fueran cero entonces la función de correlación de Pearson es el coseno entre x e y . En muchos casos se empieza entonces por centrar todas las calificaciones de los usuarios en cero, es decir restarle a cada calificación el promedio del usuario.

Para nuestro primer usuario quedaría:

$$x = (2, NA, 2, 4, 5, NA) \rightarrow \mu = 3.25 \rightarrow (1.25, NA, 1.25, 0.75, 1.75, NA)$$

Para un usuario dado, por ejemplo el u5 en nuestra matriz podemos calcular la semejanza con todos los demás usuarios usando el coseno, luego de restar el promedio a cada calificación.

$$\text{sim}(5,1) = 0.27 \quad \text{sim}(5,2) = 0.72 \quad \text{sim}(5,3) = 0.5 \quad \text{sim}(5,4) = 0.13 \quad \text{sim}(5,6) = NA$$

(suponer que estas cuentas están bien)

Ahora podemos calcular cada calificación faltante usando la siguiente fórmula:

$$r_{xi} = \frac{\sum_{y \in N} s_{xy} r_{yi}}{\sum_{y \in N} s_{xy}} \quad (17.3)$$

En donde N es el conjunto de los N usuarios más similares al que queremos estimar y r_{yi} es el rating del usuario “y” para la película “i”.

Por ejemplo podemos usar $N = 2$ para tomar los dos usuarios más similares al que nos interesa.

En el caso de la primera película del usuario 5 tenemos que el único usuario que la ha calificado y que tiene semejanza positiva con nuestro usuario es el 2. Por lo tanto la cuenta es: $(5 * 0.72)/0.72 = 5$

Es decir que estimamos que al usuario 5 le va a gustar mucho Sherlock porque Sherlock le ha gustado mucho al usuario 2 que es similar al 1.

Nota de implementación: Para tomar los “N” usuarios más parecidos al usuario al cuál le queremos recomendar podemos usar LSH que como sabemos implica tiempo sublineal.

17.4.2 Semejanza Item Item

En esta variante lo que vamos a hacer para estimar la calificación de un usuario para un ítem es buscar los ítems mas parecidos al que queremos estimar y que haya calificado el usuario. Luego estimamos la calificación haciendo un promedio ponderado entre las calificaciones del usuario para estos ítems y la semejanza que tienen con el ítem que queremos estimar.

Vamos a usar la siguiente matriz de utilidad:

		users											
		1	2	3	4	5	6	7	8	9	10	11	12
movies	1	1		3			5			5		4	
	2			5	4			4			2	1	3
	3	2	4		1	2		3		4	3	5	
	4		2	4		5			4			2	
	5			4	3	4	2				2	5	
	6	1		3		3			2			4	

 - unknown rating
 - rating between 1 to 5

Figure 17.5: Matriz de Utilidad Item Item

Supongamos que queremos estimar la calificación del usuario 5 para la película 1. El primer paso es calcular la semejanza de todas las demás películas con respecto a la uno. Para esto restamos a cada calificación el promedio de la película y usamos el coseno.

Notar que el modelo user-user restábamos a cada calificación el promedio del usuario y en el modelo ítem-ítem vamos a restar el promedio del ítem (película).

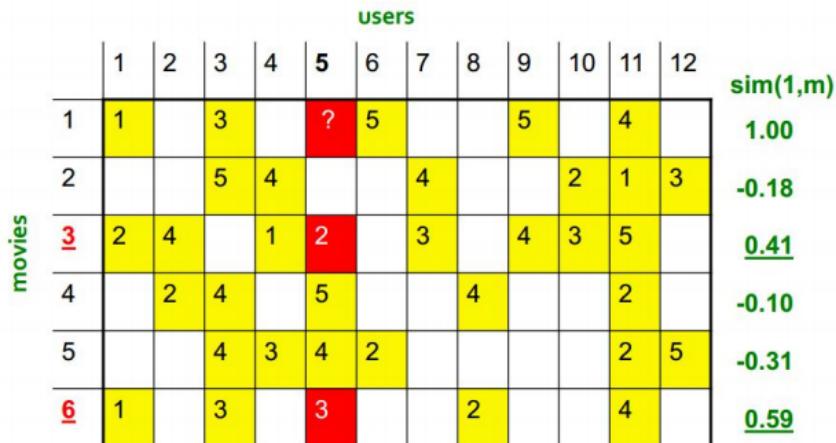


Figure 17.6: Matriz de Utilidad Ítem Ítem

Usando N=2 tomamos que las dos películas más similares a la 1 son la 6 con semejanza 0.59 y la 3 con semejanza 0.41. Por lo tanto estimamos la calificación del usuario 5 para la película 1 calculando el promedio ponderado por la semejanza:

$$(2 * 0.41 + 3 * 0.59) / (0.41 + 0.59) = 2.6$$

Nos queda:

Recordemos que podemos encontrar los “N” ítems mas similares al que queremos estimar usando LSH en tiempo sublineal.

En general el modelo ítem-ítem da mejores resultados que el modelo user-user. Esto puede explicarse pensando que los usuarios tienen gustos muy diversos y que incluso cambian con el tiempo mientras que la semejanza entre películas es algo mucho más estable para un número de calificaciones lo suficientemente grande. Aplica la ley de los grandes números!

17.4.3 Collaborative Filtering en base a Desviaciones

A continuación describimos la forma en la cual suele implementarse collaborative filtering en un sistema de recomendaciones real.

Vamos a estimar la calificación del usuario “i” a la película “j” de la siguiente forma:

$$r_{ij} = \mu + \delta_i + \delta_j + \delta_{ij} \quad (17.4)$$

	users											
	1	2	3	4	5	6	7	8	9	10	11	12
movies	1	1		3		2.6	5		5		4	
2			5	4			4			2	1	3
3	2	4		1	2		3		4	3	5	
4		2	4		5			4			2	
5			4	3	4	2				2	5	
6	1		3		3			2			4	

Figure 17.7: Matriz de Utilidad Item Item

Donde:

μ es el promedio de todas las calificaciones de todas las películas (promedio global).

δ_i es la desviación del usuario i con respecto a μ

δ_j es la desviación del ítem j con respecto a μ

δ_{ij} es la desviación del usuario i para la película j que se calcula como:

$$\delta_{ij} = \frac{\sum_{j \in N} s_{ij} (r_{ij} - b_{xj})}{\sum_{j \in N} s_{ij}} \quad (17.5)$$

Con:

$$b_{xj} = \mu + \delta_x + \delta_j \quad (17.6)$$

Veamos como funciona esto en nuestro ejemplo para el caso de que queramos estimar la calificación del usuario 5 para la película 1.

En primer lugar calculamos μ el promedio global de todas las películas: $111/35 = 3.17$

La desviación del usuario 5 es una medida de que tan severo es el usuario 5 y se calcula como la diferencia entre el promedio de calificaciones del usuario 5 y el promedio global. El promedio del usuario 5 es: $2 + 5 + 4 + 3/4 = 3.5$ Por lo tanto la desviación con respecto al promedio global es: $+0.33$

Ahora calculamos la desviación de la película 1 que mide que tan buena o mala es la película con respecto al promedio global: El promedio de la película

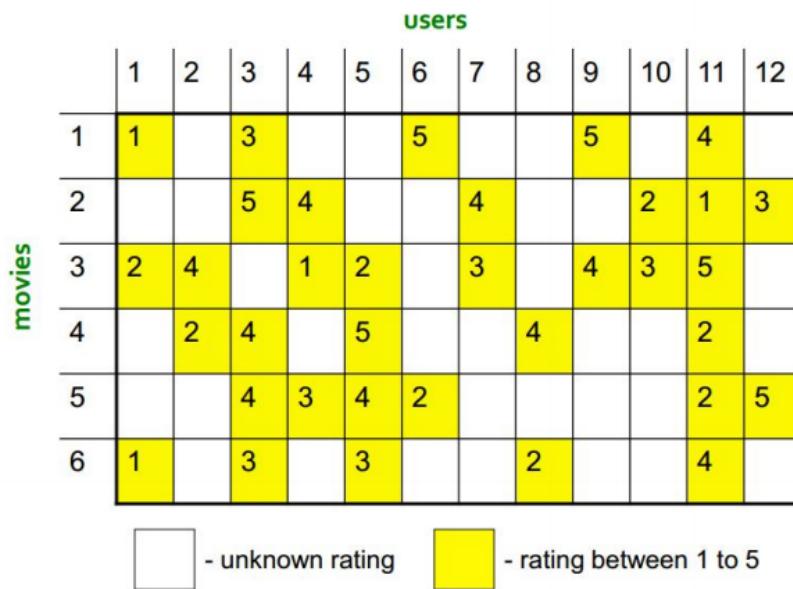


Figure 17.8: CF en base a desviaciones

1 es: $1 + 3 + 5 + 5 + 4/5 = 3.6$ Es decir que la desviación de la película 1 es: +0.43

Y ahora podemos calcular la desviación del usuario 5 con respecto a la película 1 usando la semejanza ítem-ítem pero computando el promedio ponderado de las desviaciones en lugar de las calificaciones: Habíamos calculado las 2 películas más semejantes y eran la 3 y la 6. Analicemos la desviación de la calificación del usuario 5 para la película 2.

Promedio global = 3.17

Desviación del usuario 5: +0.33

Desviación de la película 4: -0.17

Por lo tanto la desviación de la calificación "2" del usuario 5 a la película 3 es: $2 - (3.17 + 0.33 - 0.17) = -1.33$

Y para el usuario 5 y la película 6 tenemos:

Promedio global = 3.17

Desviación del usuario 5: +0.33

Desviación de la película 6: -0.57

Por lo tanto la desviación de la calificación "3" del usuario 5 a la película 6 es: $3 - (3.17 + 0.33 - 0.57) = 0.07$

Ahora calculamos el promedio ponderado de estas desviaciones de acuerdo a la semejanza ítem-ítema: $((0.41 * -1.33) + (0.59 * 0.07)) / (0.41 + 0.59) = -0.5$

Por lo tanto podemos estimar la calificación del usuario 5 película 1 como:

$$3.17 + 0.33 + 0.43 - 0.5 = 3.43$$

Observemos que esto es igual (o muy parecido) a lo que nos daba la estimación usando únicamente la semejanza ítem-ítem pero agregando la desviación del usuario y la desviación de la película. Es un usuario que suele calificar por encima del promedio y es una película que es mejor que el promedio por eso tiene sentido que hayamos pasado de 2.6 a 3.43 como estimación.

Hemos presentado el algoritmo de collaborative filtering que estima las calificaciones de los ítems en base a desviaciones a partir del promedio de todas las calificaciones. Hay una desviación propia del usuario, otra propia del ítem y finalmente otra desviación que relaciona al usuario con ese ítem y que se calcula en base a la semejanza ítem-ítem o usuario-usuario.

17.5 Evaluación de Sistemas de Recomendación

Esta sección es muy importante no sólo para poder determinar cómo evaluar un sistema de recomendaciones sino porque de aquí pueden derivarse mejoras al algoritmo que hemos visto hasta el momento.

Para construir un sistema de recomendaciones es necesario contar con un set de datos que en general consiste en la matriz de utilidad con las calificaciones de los usuarios a los distintos ítems. El objetivo es predecir de la mejor forma posible las calificaciones que faltan. Evaluar ésto no es tarea sencilla ya que necesitariamos que los usuarios nos devuelvan feedback sobre las recomendaciones que hemos realizado, sería deseable que el sistema pueda realizar recomendaciones buenas sin necesitar retro-alimentarse de la opinión de los usuarios sobre las recomendaciones del mismo.

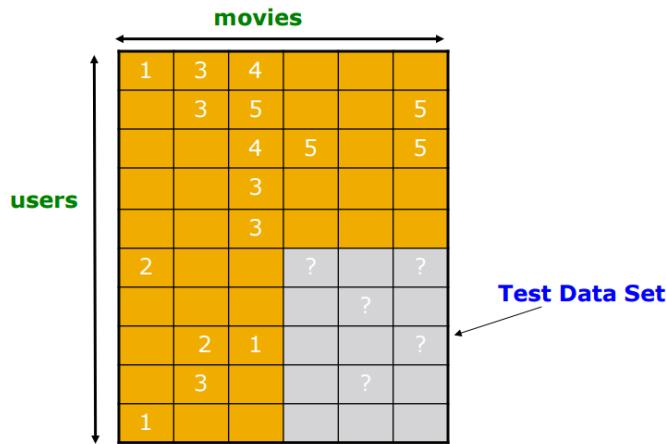


Figure 17.9: Evaluación de Sistemas de Recomendaciones

La solución a este problema es bastante simple y consiste en separar un cierto porcentaje de calificaciones al azar, por ejemplo un 20% del total. Con el sistema de recomendaciones y luego lo probamos realizando predicciones para las calificaciones que hemos removido. Como conocemos el valor real de estas calificaciones podemos calcular la efectividad del sistema mediante un simple cálculo:

$$J = \frac{1}{N} \sqrt{\sum_{ij} (\hat{r}_{ij} - r_{ij})^2} \quad (17.7)$$

A esta métrica se la denomina RMSE (root mean squared error), también puede calcularse sin la raíz en cuyo caso se denomina SSE.

Gracias a esta métrica podemos probar diferentes algoritmos y diferentes parámetros para nuestros algoritmos de forma tal de encontrar el algoritmo que minimice el error total. Es recomendable realizar varios tests tomando cada vez un 80% y 20% al azar de las calificaciones y calcular el promedio de error de todas las pruebas. De esta forma evitamos que un cierto muestreo de las calificaciones sea muy influyente en cuanto al rendimiento del sistema. A esta técnica en aprendizaje automático se la llama “cross validation”

17.6 Modelos Latentes

Las calificaciones que los usuarios realizan de los ítems, en nuestro caso películas, aportan valiosa información sobre las preferencias de los mismos. Por ejemplo podemos observar que un usuario que ha calificado muy bien a Star Wars, Star Trek y Superman probablemente sea fanático de la ciencia ficción y por lo tanto deberíamos estimar una calificación alta para ese tipo de películas.

El algoritmo de collaborative filtering se limita únicamente a calcular semejanzas entre películas pero no es capaz de darse cuenta que un grupo de películas similares expresa la actitud del usuario hacia las películas de dicho tipo. A este tipo de modelización la llamamos “modelo latente” porque nos permite inferir preferencias de los usuarios que no están explícitas en la matriz de utilidad.

Hay dos métodos que funcionan muy bien para obtener la información latente de la matriz de utilidad: RBMs o SVD++

17.6.1 SVD++

El algoritmo SVD++ [Koren] fue presentado informalmente durante la competencia del premio Netflix por un millón de dólares. Este algoritmo representó un importante salto en la mejora de los sistemas de recomendaciones, hoy en día se usa extensamente. Está basado en el simple concepto de factorizar la matriz de utilidad para descubrir los modelos latentes.

El principio aquí es muy simple: descomponer la matriz de utilidad en el producto de dos matrices: Q y P. En donde Q tendrá tantas filas como ítems y tantas columnas como factores latentes usemos y P tendrá tantas filas como usuarios y tantas columnas como factores.

$$U_{mxn} = Q_{mxr}P_{nrx}^t \quad (17.8)$$

La cantidad de factores “r” es un parámetro que debemos fijar. Al descomponer la matriz Q nos dará la relación entre los ítems y los factores y P nos dará la relación entre los ítems y los factores. Es decir que en Q vamos a poder observar que tipo de factores están asociados a cada película y en P tendremos las preferencias de los usuarios con respecto a los factores.

El objetivo es encontrar Q y P tales que $Q * R$ minimice el error con la matriz de utilidad original.

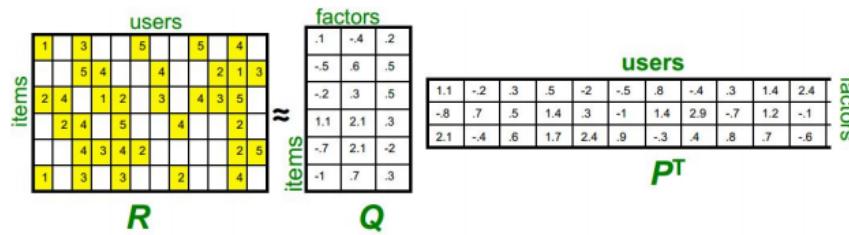


Figure 17.10: Factorización de la Matriz de Utilidad

Si logramos hallar estas matrices Q y P entonces podemos estimar la calificación del usuario i para la película j haciendo $q_i * p_j$. Notar que para las calificaciones que ya teníamos esto debería darnos un número muy similar al de la matriz de utilidad ya que hemos construido a P y Q para minimizar la diferencia entre QP y la matriz original. Lo interesante de esto es que las matrices Q y P que minimicen el error con la original nos dan mágicamente todos los valores que nos faltaban en la matriz de utilidad original.

Por ejemplo en nuestro caso si queremos estimar la calificación del usuario 5 para el ítem 2. Hacemos $q_2 = (-0.5, 0.6, 0.5)$ y $p_5 = (-2, 0.3, 2.4)$ $q_2 * p_5^t = 2.4$

Es decir que estimamos una calificación de 2.4

Tenemos que ahora que analizar de que forma hallar las matrices Q y P .

Notamos que el problema es muy similar a la indexación semántica latente en donde usamos la descomposición SVD para reducir las dimensiones de una matriz y obtener un modelo latente. Desafortunadamente no podemos usar SVD para nuestro sistema de recomendaciones porque la matriz de utilidad no está completa, tiene muchos “huecos” SVD no funciona si no tenemos todos los datos de la matriz original. Necesitamos por lo tanto un algoritmo similar a SVD pero que funcione con matrices incompletas. Algo así como una SVD parcial.

Podemos escribir el siguiente problema de optimización:

$$\min_{P,Q} \sum_{(i,x) \in R} (r_{xi} - q_i p_x)^2 \quad (17.9)$$

En donde la matriz r que usamos es nuestro set de entrenamiento, es decir el 80% de los datos de nuestro sistema.

Aquí tenemos que detenernos a pensar porque pese a que aún no tenemos el algoritmo ya tenemos un potencial problema con el mismo. Podríamos encontrar valores para las matrices Q y P que minimicen la diferencia con la matriz de utilidad de nuestro set de entrenamiento pero que funcionen muy mal para el set de pruebas (el otro 20% que reservamos para evaluar el sistema). Esto es un clásico problema de overfitting en el cual nuestro algoritmo aprende a ajustar muy bien al set de entrenamiento pero funciona muy mal para realizar predicciones.

La forma de solucionar el overfitting es intentar minimizar las normas de las matrices Q y P . A esto se lo llama “regularización”.

Nuestro problema es entonces:

$$\min_{P,Q} \sum_{(i,x) \in R} (r_{xi} - q_i p_x)^2 + [\lambda_1 \sum_x \|p_x\|^2 + \lambda_2 \sum_o \|q_i\|^2] \quad (17.10)$$

Donde λ_1 y λ_2 son hiper-parámetros a definir.

Como en tantos otros problemas de optimización podemos encontrar la solución mediante el método de Gradient Descent (GD). Concretamente usaremos la variante “Stochastic gradient descent” en donde vamos a ir optimizando P y Q en base a cada una de las calificaciones, que ya conocemos, procesando éstas calificaciones una por una.

Algoritmo de Gradient Descent para calcular Q y P

Algorithm 41: SVD++

- 1 Inicializar Q y P con los valores de la SVD suponiendo que las calificaciones que faltan son cero.
 - 2 **while** *not convergencia* **do**
 - 3 **for** $r_{ij} \in R$ **do**
 - 4 Actualizar $Q_{ij} = Q_{ij} - \alpha \nabla Q_{ij}$
 - 5 Actualizar $P_{ij} = P_{ij} - \alpha \nabla P_{ij}$
-

Calcular el gradiente para Q y P para una calificación r_{xi} es sencillo:
La derivada del error es: $\epsilon_{xi} = r_{xi} - Q_i P_x$ Por lo tanto:

$$\begin{aligned} Q_i &= Q_i + \alpha(\epsilon_{xi} P_x - \lambda_1 Q_i) \\ P_x &= P_x + \alpha(\epsilon_{xi} Q_i - \lambda_2 P_x) \end{aligned} \quad (17.11)$$

Este algoritmo es eficiente y funciona muy aceptablemente. Por lo que hemos obtenido una solución muy eficiente para estimar las calificaciones que nos faltan

en nuestro sistema en base al modelo latente. Como se podrán imaginar estamos muy lejos de haber terminado.

Notar que los parámetros lambda no se optimizan en el algoritmo. A los parámetros que no se optimizan dentro del algoritmo los llamamos “hiper-parámetros”. Los hiper-parámetros en general se buscan realizando un grid-search en base a una serie de valores posibles. Por ejemplo podemos probar con $\lambda = \{0.005, 0.05, 0.5, 0, 0.5, 1.2\}$ y entonces tenemos 36 combinaciones a probar para nuestros dos valores de lambda. Corremos el algoritmo de optimización SGD para cada valor y nos quedamos con los lambda que minimicen el error.

Agregando las desviaciones del usuario y de la película.

Podemos volver a pensar que es conveniente modelar la recomendación en base a desviaciones a partir del promedio de calificaciones global:

$$r_{xi} = \mu + b_x + b_i + q_i p_x \quad (17.12)$$

Sabemos como calcular Q y P , ahora QP nos dará una desviación en lugar de una recomendación pero el algoritmo es exactamente el mismo. μ es el promedio global de todas las calificaciones. Y b_x y b_i ahora son valores que podemos también aprender en lugar de calcularlos. Es decir que buscaremos los b_x y los b_i que mejor ajusten a nuestro sistema.

Escribimos el problema de optimización:

$$\min_{Q,P} \sum_{(x,i) \in R} (r_{xi} - (\mu + b_x + b_i + q_i p_x))^2 + \lambda_1 \sum_i \|q_i\|^2 + \lambda_2 \|p_x\|^2 + \lambda_3 \sum_x \|b_x\|^2 + \lambda_4 \sum_i \|b_i\|^2$$

Y lo resolvemos por SGD pero ahora ademas de actualizar Q y P tenemos también que ir actualizando b_x y b_i en cada paso. Lo cual no representa ningún problema pues solo implica hacer un par de actualizaciones mas en el código.

17.7 Factores Temporales

Habíamos mencionado, ya hace un buen tiempo, en este mismo apunte que los factores temporales son muy importantes en un sistema de recomendaciones.

Afortunadamente es sencillo incorporar el tiempo en nuestro algoritmo. Los parámetros que se ven afectados por el tiempo son b_x y b_i es decir la desviación del usuario y la desviación de la película. Los usuarios cambian de opinión a lo largo del tiempo y la percepción de una película también cambia a lo largo del tiempo.

Lo que hacemos es dividir nuestro set de entrenamiento (el 80% de los datos) en intervalos de tiempo de una cierta duración fija, por ejemplo 10 semanas, vamos a tener entonces tantos valores para b_x y b_i como intervalos de tiempo tengamos en total. Al ir optimizando los parámetros vamos a usar el b_x y el b_i que correspondan según la fecha en que fue realizada cada calificación.

Planteamos entonces

$$r_{xi} = \mu + b_x(t) + b_i(t) + q_i + p_x \quad (17.13)$$

17.8 Factorization Machines

El algoritmo FM [Rendle] es uno de los que mejores resultados suele dar en competencias en las cuales hay que construir un sistema de recomendaciones, este algoritmo puede usarse tanto para recomendaciones como para clasificación o regresión y se basa en encontrar los factores latentes asociados a cada variable del sistema.

Para empezar la presentación del algoritmo supongamos que tenemos calificaciones de usuarios en triples del estilo {userid,movieid,rating}. Podríamos pensar en usar una simple regresión lineal para estimar la calificación de un usuario a una película de la forma:

$$\hat{y} = w_0 + w_1 \text{userid} + w_2 \text{movieid}$$

Por supuesto que esto no funcionaría ya que no tiene sentido usar el userid y movieid como variables numéricas en una regresión lineal, que el userid sea 10 o 20000 no debería afectar el coeficiente para estimar la calificación del ítem. La solución a este problema es convertir el userid y movieid en "n" y "m" variables binarias mediante "one-hot encoding" si tenemos 4 usuarios el usuario 2 quedaría codificado como (0,1,0,0) y si tenemos 5 películas la película 3 quedaría (0,0,1,0,0) y entonces podríamos tener vectores de la forma: (0,1,0,0,0,0,1,0,0,4) indicando que el usuario 2 de 4 calificó con 4 la película 3 de 5.

Esto nos lleva a un modelo de regresión lineal de la forma:

$$\hat{y} = w_0 + \sum_{i=1}^n w_i * x_i + \sum_{j=1+n}^{m+n} w_j * x_j$$

Este modelo captura un "coeficiente" por cada una de las variables del modelo y estima la calificación en base a estos coeficientes. En otras palabras la calificación va a depender del usuario y la película. Es muy similar al modelo en el cuál usábamos la desviación del usuario y la película para estimar la calificación. Podemos extender este modelo agregando otras variables codificadas mediante one-hot encoding como por ejemplo cuáles son otras películas calificadas por el usuario, en que momento hizo la calificación o cuál es la última película calificada.

En la figura 17.11 podemos observar como quedarían los vectores, notemos que en un sistema real estos vectores tienen cientos de miles o millones de dimensiones ya que dependen de la cantidad de usuarios e ítems en el sistema.

Con este modelo podríamos crear una regresión lineal usando todos los features, este modelo sin embargo no capturaría la interacción entre las diferentes variables, por ejemplo tendríamos un coeficiente para el usuario, otro para la película pero no tendríamos un coeficiente que relacione el usuario o la película. Al "factorizar" lo que hacemos es agregar todos los coeficientes de interacción

Feature vector x											Target y			
$x^{(1)}$	1	0	0	...	1	0	0	0	...	0.3	$y^{(1)}$			
$x^{(2)}$	1	0	0	...	0	1	0	0	...	0.3	$y^{(2)}$			
$x^{(3)}$	1	0	0	...	0	0	1	0	...	0.3	$y^{(3)}$			
$x^{(4)}$	0	1	0	...	0	0	1	0	...	0	$y^{(4)}$			
$x^{(5)}$	0	1	0	...	0	0	0	1	...	0.5	$y^{(5)}$			
$x^{(6)}$	0	0	1	...	1	0	0	0	...	0.5	$y^{(6)}$			
$x^{(7)}$	0	0	1	...	0	0	1	0	...	0.5	$y^{(7)}$			
A	B	C	...	T1	NH	SW	ST	...	T1	NH	SW	ST	...	$y^{(8)}$
User	Movie	Other Movie rated	Time	Last Movie rated										

Figure 17.11: Features codificados mediante One-Hot-Encoding para armar un Factorization Machine

entre cada una de las variables del modelo en total agregamos $\binom{n_2}{n_1}$ nuevas variables de tipo w_{ij} indicando la relación entre la variable i y la variable j del modelo.

$$\hat{y} = w_0 + \sum_{i=1}^n w_i * x_i + \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} * x_i * x_j$$

Este modelo es muy completo pero tiene como problema que la cantidad de coeficientes sufre de un problema de explosión combinatoria, teniendo millones de usuarios y millones de películas no hay forma de manejar todas las interacciones posibles entre las variables del modelo. Este es el punto en el cuál el algoritmo de Factorization Machines aporta una solución muy ingeniosa y elegante.

La idea es representar a cada variable (columna) de nuestro modelo mediante un vector de "k" elementos siendo "k" la cantidad de factores latentes que queremos usar. Si hacemos esto entonces podemos re-escribir nuestro modelo de regresión de la forma:

$$\hat{y} = w_0 + \sum_{i=1}^n w_i * x_i + \sum_{i=1}^n \sum_{j=i+1}^n \langle v_i, v_j \rangle * x_i * x_j \quad (17.14)$$

La cantidad total de parámetros a estimar ahora es: w_0 , "n" coeficientes w_i y "n" vectores v_i de k dimensiones cada uno. Este número de parámetros es perfectamente manejable incluso en sistemas con millones de usuarios e ítems.

17.8.1 Expresividad de FM

Notemos que los parámetros de interacción w_{ij} están ahora factorizados mediante el producto interno $\langle v_i, v_j \rangle$ con lo cuál seguimos teniendo la interacción entre todas las variables del modelo pero factorizadas de acuerdo a k factores latentes. k es un hiper-parámetro que podemos buscar por grid-search como es costumbre.

Sabemos que para un k suficientemente largo existe una matriz V de forma tal que $W = VV^t$, esto quiere decir que el modelo tiene el poder de expresar todas las interacciones entre las variables que definimos. Al restringir la dimensionalidad de los vectores V a k dimensiones evitamos el overfitting ya que hacemos que el modelo tenga que aproximar W mediante una matriz de rango k .

17.8.2 Como funcionan los parámetros de una FM

Supongamos que tenemos los siguientes datos:

```

U = {Alice (A), Bob (B), Charlie (C), . . .}
I = {Titanic (TI), Notting Hill (NH), Star Wars (SW),
Star Trek (ST), . . .}

S = {(A, TI, 2010-1, 5),(A, NH, 2010-2, 3),(A, SW, 2010-4, 1),
(B, SW, 2009-5, 4),(B, ST, 2009-8, 5),
(C, TI, 2009-9, 1),(C, SW, 2009-12, 5)}

```

Queremos estimar la interacción entre Alice (A) y Star Trek (ST) para predecir el rating. No tenemos ningún caso en el cual "A" haya calificado "ST" por lo que hace falta una estimación/predicción. La estimación se puede calcular mediante la interacción de los vectores vA y vST . En primer lugar vemos que Bob y Charlie van a tener vectores vB y vC similares porque ambos tienen interaccion similar con Star Wars (vSW) por lo tanto $\langle vB, vSW \rangle$ y $\langle vC, vSW \rangle$ tienen que ser similares. Alice (vA) va a tener un vector diferente al de Charlie (vC) porque difieren en la calificación de Titanic y Star Wars. Finalmente los vectores de Star Trek vST y Star Wars vSW tienen que ser similares porque Bob tienen calificaciones similares para ambos. En total esto quiere decir que el producto interno entre los vectores de Alice y Star Trek tiene que ser similar al producto interno entre Alice y Star Wars lo cual tiene sentido.

17.8.3 Algoritmos basados en FMs

Podemos usar Factorization Machines para regresión como hemos visto hasta ahora, lo cuál nos sirve en un sistema de recomendaciones para estimar la calificación de cualquier usuario para cualquier ítem.

Para clasificación podemos usar $y \in [1, 0]$ como variable destino y la función sigmoida para estimar a que clase pertenece cada vector.

Finalmente para rankings podemos usar pares de vectores (x_a, x_b) indicando en la variable destino y si uno es mayor que el otro o no, esto nos construye una función de comparación que luego podemos usar para ordenar cualquier conjunto de vectores para los cuales no conocemos el valor de y

17.8.4 Aprendizaje de FMs

El aprendizaje de una FM es muy sencillo y puede realizarse por varios métodos como por ejemplo "Stochastic Gradient Descent" (SGD). El gradiente del modelo es 1 para w_0 , x_i para w_i y $x_i \sum_{j=1}^n v_j x_j - v_i x_i^2$ para v_i .

La implementación de referencia de FMs es LibFM y es muy simple de usar desde cualquier lenguaje de programación.

17.8.5 LibFM

El formato de LibFM es casi igual al de LibSVM, cada línea contiene un vector de nuestro set de datos, por cada vector indicamos primero el label (clase o rating) y luego pares de tipo columna:valor ya que los vectores son siempre muy dispersos. LibFM entrena en base a un archivo de entrenamiento en este formato y luego predice los valores para un archivo de test que tiene igual formato pero para el cual los labels se ignoran (pueden ser todos ceros).

A modo de ejemplo usaremos 4 películas (dimensiones 0 a 3) y 5 usuarios (dimensiones 4 a 8), las películas 0 y 1 son de una misma hipotética categoría y las películas 2 y 3 también. Usaremos entonces 2 factores latentes ($k = 2$).

El archivo de entrenamiento es:

```
5 0:1 4:1
5 1:1 4:1
1 2:1 4:1
2 3:1 4:1
5 2:1 5:1
5 3:1 5:1
1 0:1 5:1
5 0:1 6:1
5 1:1 6:1
1 2:1 6:1
1 3:1 6:1
5 2:1 7:1
5 3:1 7:1
1 1:1 7:1
5 0:1 8:1
1 2:1 8:1
```

Y el archivo de test es:

```
0 1:1 8:1
0 3:1 8:1
```

Es decir que para el usuario 8 a cual le gustó la película cero y no le gustó la 2 queremos predecir la calificación para las películas 1 y 3. El comando que usamos es:

```
libfm -task r -method mcmc -train train.libfm -test test.libfm -init_stdev 0.5 -iter 100 -dim '1,
```

init_stdev es un hiper-parámetro del cual depende la velocidad de convergencia por lo que hay que elegirlo con cuidado. Con la opción -dim elegimos si usamos un bias, si usamos interacciones de 1 nivel y el valor de k, en nuestro caso $k = 2$

Y el resultado obtenido es:

```
4.37187
1.86926
```

Notemos que esto está de acuerdo a lo esperado, al usuario le debería gustar la película 1 pero no la película 3.

El algoritmo Factorization Machines es uno de los mas poderosos para sistemas de recomendaciones ya que tiene mayor poder expresivo que otros algoritmos basados en factores latentes pero al mismo tiempo tiende a evitar el overfitting gracias al uso de la factorización. Sin dudas es uno de los algoritmos mas importantes de este capítulo.

17.9 El Problema del Cold-Start: Multi-Armed Bandits

Uno de los problemas mas difíciles en un sistema de recomendaciones es como incorporar nuevos ítems al sistema, para los ítems que son populares podemos no hacer nada y eventualmente estos van a tener suficientes rankings como para ser recomendados, pero esto es algo que a un sistema de recomendaciones no le importa porque los ítems populares siempre van a ser consumidos, lo importante en un sistema de recomendaciones es poder recomendar ítems que no sean populares y que puedan ser del agrado de los usuarios, esto maximiza el uso del "long tail". Los ítems que son nuevos y no son populares (masivos) nunca son recomendados y por lo tanto nunca reciben rankings lo cual da genera un círculo vicioso que conocemos como *cold start problem*

El objetivo es entonces desarrollar un algoritmo que nos permita recomendar ítems nuevos a los usuarios sin que estos sean populares. Algunos de estos ítems van a ser buenos y otros no, incluso pueden ser muy malos. El problema es similar a jugar con máquinas tragamonedas en donde sabemos que alguna máquina da mas premios que las otras pero no sabemos cual, tenemos que descubrir esto mediante experimentación, es por esto que los algoritmos que se encargan de este tipo de cuestiones se denominan "*multi-armed bandits*"

17.9.1 Multi-Armed Bandits

Supongamos que medimos si a un usuario le gustó o no una recomendación mediante un 1 o 0, supongamos que tenemos varios ítems y decidimos usar un algoritmo greedy es decir recomendar el ítem que tiene mejores resultados hasta ahora. Ofrecemos el ítem k_1 a un usuario cualquiera y al usuario no le gusta $x_1 = 0$ ofrecemos el ítem k_2 a algún usuario y al usuario le gusta $x_2 = 1$, a

partir de aquí el ítem k_1 no va a volver a ser recomendado nunca más ya que tiene resultado $=0$ y el ítem k_2 en el peor de los casos será $1/N > 0$. Greedy no funciona, porque abandona rápidamente a los ítems que no fueron exitosos en su primer intento.

La clave consiste en combinar *exploración* y *explotación*. La explotación consiste en recomendar la mejor alternativa en base a los resultados observados. La exploración consiste en ofrecer nuevas alternativas. El algoritmo greedy explota pero no explora.

17.9.2 Algoritmo Epsilon

En este algoritmo en cada instante con probabilidad ϵ elegimos el ítem con mejor promedio hasta ahora y con probabilidad $1 - \epsilon$ elegimos un ítem cualquiera al azar (de los ítems que son nuevos obviamente).

El hiper-parámetro ϵ controla el balance entre exploración y explotación y el valor óptimo depende de los datos. Uno de los problemas del algoritmo epsilon es que la exploración no tiene criterio, es completamente aleatoria.

17.9.3 Algoritmo UCB1

En cada instante "t" elegimos el mayor valor de $UCB(j)$.

$$UCB(j) = \hat{\mu}_j + \alpha \sqrt{\frac{2 \ln(t)}{\text{tries}(j)}} \quad (17.15)$$

Donde $\text{tries}(j)$ es la cantidad de veces que intentamos el ítem j , $\hat{\mu}_j$ es el promedio de éxito del ítem j hasta el momento. α es un hiper-parámetro.

El promedio de cada ítem se puede estimar mediante algún algoritmo de Machine Learning (regresión) por ejemplo en base a los features propios de cada ítem, como su director, actores, género, presupuesto, etc. Este promedio puede usarse como μ_j para UCB1.

Los algoritmos que vimos en esta sección no solo se aplican al problema del cold-start en sistemas de recomendaciones sino a muchos otros casos, como por ejemplo colocación de avisos en páginas web en base a su CTR (click through rate), posición de los artículos en un sitio de noticias, etc.

Y a propósito si alguien quiere hacerle daño a una empresa que publicita en la web lo peor que podemos hacerles es clickear en sus avisos, esto no solo hace que se les cobre sino que genera un aumento en la probabilidad de que estos avisos sean mostrados generando un gasto todavía mayor.

17.10 Diversidad: Optimización Submodular

Uno de los puntos importantes en un sistema de recomendaciones es realizar sugerencias que abarquen una amplia selección de distintas categorías para los ítems. No queremos recomendarle al usuario películas o libros que sean todas del mismo género.



Figure 17.12: Diversidad

Vamos a suponer que tenemos una cierta cantidad de ítems que podemos recomendar y que cada ítem pertenece a una o varias categorías. El objetivo es que los ítems que vamos a recomendar cubran la mayor cantidad de temas posibles. Este problema es conocido, se llama *set-cover* y lamentablemente es NP-completo es decir que no existe una solución en tiempo polinómico. Vamos a suponer que la cantidad de ítems que vamos a recomendar es k

Una aproximación es usar un algoritmo greedy:

Algorithm 42: Diversidad by Greedy

```

1  $A_0 = \{\}$ 
2 for  $i = 1..k$  do
3   elegir el ítem cuyo conjunto de categorías  $d$  maximiza:  $F(A_{i-1} \cup d)$ 
4    $A_i = A_{i-1} \cup d$ 
  
```

Es fácil probar que greedy no es óptimo: Sea $S_1 = \{1, 2, 3, 4, 5\}, S_2 = \{1, 2, 3, 7\}, S_3 = \{4, 5, 6, 8\}$. Greedy toma en primer lugar a S_1 que nos agrega 5 categorías, luego tomamos a S_3 que nos agrega dos categorías (6 y 8) contra S_2 que solo agregaría 7. Finalmente agregamos S_2 . Como podemos ver esto cubre todas las categorías usando 3 ítems (tres conjuntos) pero en realidad los conjuntos correspondientes a los ítems 2 y 3 (S_2 y S_3) cubrirían todas las categorías por lo que sería una mejor solución.

Se puede demostrar que Greedy aproxima al menos un 63% la solución óptima cuando la función a optimizar es submodular. Esto ya lo hemos usado en redes sociales para maximizar influencia.

Recordemos que una función $F(S)$ es submodular si agregar un elemento al conjunto S nos da una ganancia en F que es peor o igual a agregar ese mismo elemento a un subconjunto de S

$$\begin{aligned}
 & A \subseteq B \\
 & F(A \cup d) - F(A) \geq F(B \cup d) - F(B)
 \end{aligned} \tag{17.16}$$

Intuición: Si agregamos un mismo conjunto nuevo a un subconjunto de S

seguramente la ganancia es mayor a agregarlo a S (porque S cubre más cosas que cualquiera de sus subconjuntos)

La submodularidad es la versión discreta de la concavidad!

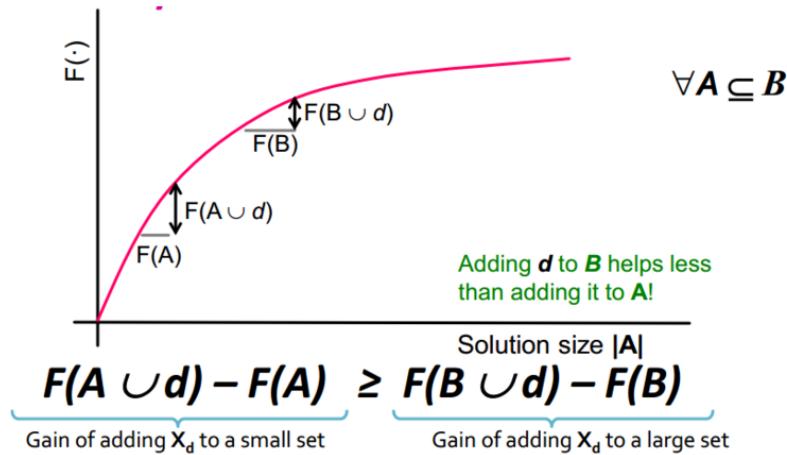


Figure 17.13: Submodularidad

Es fácil probar que en el caso de la diversidad la función F es submodular. Por lo tanto sabemos que el algoritmo greedy es en el peor caso un 63% de la solución óptima. Es decir que $F(A) > 0.63 * \text{OPT}$. El problema de greedy es que es muy lento. En cada iteración tenemos que evaluar la ganancia marginal que nos daría agregar cada uno de los ítems a nuestro conjunto de recomendaciones.

Podemos optimizar nuestro algoritmo greedy mediante una versión lazy que llamaremos, por supuesto, *lazy greedy*. En esta versión llevaremos una lista ordenada de la ganancia que nos daría cada ítem, elegimos el ítem de mayor ganancia de esta lista y lo agregamos al conjunto de recomendaciones, ahora quitamos ese ítem de la lista y recalculamos la ganancia solamente el ítem que queda con mayor ganancia (o los ítems si hay empate). Esto es válido por definición de submodularidad: $\Delta_i(d) \geq \Delta_j(d)$ para $i < j$ porque $A_i \subseteq A_j$ los márgenes $\Delta_i(d)$ solo pueden disminuir a medida que la cantidad de iteraciones i aumenta.

Veamos el siguiente ejemplo:

```
s1={1,3,6}
s2={2,3,4,6}
s3={1,4,7}
s4={3,9}
s5={2,3,4}
s6={1,9}
s7={5,7,8}
s8={2,5,8}
```

La ganancia es:

```
s1 = 3
s2 = 4
s3 = 3
s4 = 2
s5 = 3
s7 = 3
s8 = 3
...
...
```

Elegimos entonces s_2 como nuestro primer conjunto: $\{2, 3, 4, 6\}$ ahora recalculamos la ganancia solo para los que quedan con ganancia máxima (3) luego de eliminar s_2 .

```
s1 = 2
s3 = 2
s4 = 2
s5 = 1
s7 = 2
s7 = 2
s8 = 1
...
...
```

De estos elegimos s_1 , nuestro conjunto es ahora $s_1 \cup s_2$ y recalculamos:

```
s3 = 1
s4 = 1
s5 = 1
s7 = 2
s8 = 1
...
...
```

Ahora el ganador es S_7 . etc...

Podemos ver que Greedy es muchísimo mas rápido que la búsqueda del óptimo por fuerza bruta (que es exponencial) y que lazy-greedy es aun más rápido que greedy. Recordemos que el resultado de lazy-greedy es idéntico al de greedy por submodularidad de la función F .

17.10.1 Recomendaciones Personalizadas

Es lógico suponer que a un usuario pueden gustarle mas ciertas categorías que otras, es decir que cada usuario puede tener un vector de pesos indicando sus preferencias por cada una de las categorías a las que puede pertenecer un ítem.

Nuestro objetivo es ahora maximizar $w * s$ donde s es un vector binario según el tema esté cubierto o no por los ítems recomendados y w es el vector de pesos

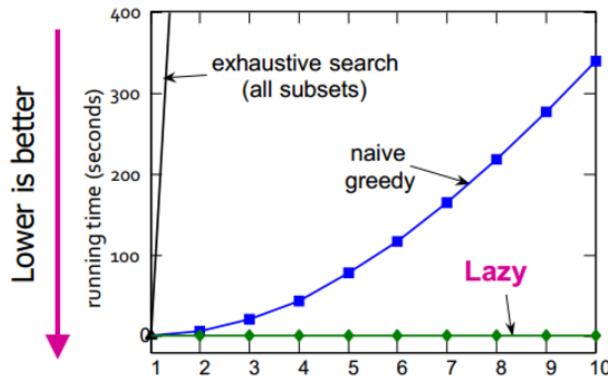


Figure 17.14: Lazy Greedy

del usuario. Afortunadamente lazy greedy funciona perfectamente. Realizamos la recomendación usando lazy-greedy con la fórmula de $w * s$ para estimar la ganancia marginal de cada ítem. Luego el usuario nos va a decir si el ítem recomendado le gustó o no y usamos este feedback para actualizar el vector w

$$w_i = \beta^r * w_i \quad (17.17)$$

Si el feedback es positivo entonces multiplicamos W_i por β si el feedback es negativo (-1) entonces dividimos W_i por beta.

17.11 Learning to Rank

El paso final de todo sistema de recomendaciones es decidir en qué orden presentar las recomendaciones al usuario. Las calificaciones que hemos pronosticado nos ayudan a saber qué ítems recomendar pero entre un ítem para el cual pronosticamos 4.64 y otro para el cual pronosticamos 4.63 no queda muy claro en qué orden deberíamos mostrarlos. Los dos son prácticamente iguales en cuanto a la probabilidad de que al usuario le interese el ítem.

La solución es usar algoritmos de *Learning to Rank* como los que hemos visto en el capítulo 7 para aprender automáticamente el orden en el que debemos presentar los ítems a los usuarios.

17.12 El Sistema Completo

Un sistema de recomendaciones tiene muchos componentes, necesitamos recolectar la mayor cantidad posible de calificaciones para poder alimentar el algoritmo

de collaborative filtering, en muchos sistemas modernos no contamos con calificaciones sino con información sobre que ítem compró cada usuario (o consumió). Mediante collaborative filtering podemos predecir si un usuario va a comprar o consumir cada uno de los ítems que no ha consumido aun.

Collaborative filtering es un proceso costoso en donde uno de los puntos críticos es la búsqueda de ítems semejantes, para esta tarea podemos usar LSH como una forma de aproximar de forma eficiente el problema de los vecinos mas cercanos. El uso de factores latentes como la SVD++ o RBMs mejora los resultados de Collaborative filtering notablemente.

CF nos da una lista de ítems que pensamos pueden gustarle al usuario, de esta lista tenemos que filtrar aquellos que por su popularidad resultan "obvios" (serendipity) y podemos aplicar un algoritmo de diversidad para que los resultados abarquen la mayor cantidad de categorías posibles.

Finalmente mediante Learning to Rank ordenamos las recomendaciones y se las entregamos al usuario.

Hay muchos detalles mas que son particulares de cada sistema, estos sistemas son tremadamente importantes en el mundo moderno generando en algunos casos mas del 50% de los ingresos de empresas como Netflix o Amazon que necesitan de buenas recomendaciones para generar mas ventas.

Chapter 18

Análisis Topológico de Datos

Data has shape and shape has meaning

- Gunnar Carlson

El análisis topológico de datos es una ciencia nueva que aplica los conceptos de topología algebraica al análisis de datos. La topología es una rama del Álgebra que se dedica a estudiar la forma de los objetos de forma independiente de sus características geométricas, como por ejemplo sus coordenadas. Un ejemplo muy usado es el que compara una taza de café con una dona o rosquilla:

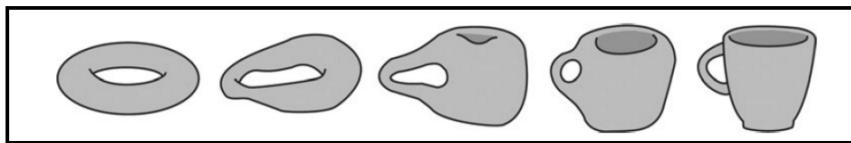


Figure 18.1: Topología

Desde el punto de vista de la topología una dona y una taza de café son la misma cosa, es decir que son un “homomorfismo” ya que existe una deformación continua que permite transformar una dona en una taza y esta transformación tiene una inversa que también es continua.

Para estudiar los datos desde el punto de vista topológico es necesario ver a los datos como un objeto con forma y para esto es necesario definir una métrica que nos permita obtener la distancia entre dos puntos cualesquiera de nuestro set de datos. Esta métrica puede ser cualquier distancia y es fundamental ya que de ella depende todo el análisis que podemos hacer de nuestros datos.

Algunas métricas posibles que hemos usado a lo largo del texto:

- distancia euclideana
- distancia Manhattan
- Norma l_0
- Norma l_∞
- Distancia de Mahalanobis
- Distancia de edición
- Distancia Normalizada de Compresión (NCD)
- Distancia de Jaccard
- Distancia coseno

18.1 Propiedades del Análisis Topológico

Hay tres propiedades fundamentales del análisis topológico de datos:

- Independencia del sistema de coordenadas.
- Invariancia a deformaciones
- Representación compacta de los datos

En el transcurso del capítulo van a quedar más claros estos tres conceptos pero intentemos una breve explicación preliminar.

La independencia del sistema de coordenadas establece que desde el punto de vista topológico las coordenadas en las cuales residen los puntos no son realmente importantes sino que importa la forma que tienen nuestros datos y para la forma lo importante es la distancia entre los puntos y no su ubicación.

La invariancia a las deformaciones establece que si aplicamos una deformación continua e inversible a nuestros datos el resultado es topológicamente idéntico al original. Esto se denomina “homomorfismo”. Un hexágono y un círculo son homomórficos ya que es posible aplicar una deformación continua para pasar de uno a otro y viceversa.

La representación compacta implica que vamos a mostrar nuestros datos como un grafo en donde la cantidad de nodos y aristas será significativamente menor a la cantidad de puntos en nuestro set de datos. Esto se ve claramente en la equivalencia topológica entre el círculo y el hexágono. Un hexágono está formado por 6 puntos y 6 lados y es topológicamente idéntico a un círculo que tiene infinitos puntos.

18.2 Los Datos como un Grafo

A la topología le interesa la "forma" que tienen los datos y no otras propiedades, por ejemplo las distancias entre los puntos no es algo importante, una esfera de radio 1 o de radio 1 millón tienen exactamente las mismas propiedades topológicas. Es parte de la invariancia a las deformaciones. Para estudiar la forma de los datos tenemos que construir un *complejo simplicial* a partir de los mismos.

Un complejo simplicial es similar a un grafo pero con algunas diferencias importantes, los componentes no son puntos y aristas sino simplices n-dimensionales.

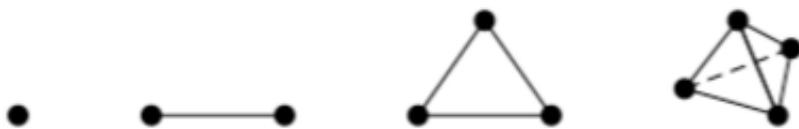


Figure 18.2: Símplices

Un solo punto forma un simplex de 0 dimensiones. Dos puntos forman un segmento, un simplex de una dimensión. Tres puntos forman un triángulo, que es un simplex de 2 dimensiones. Aquí debemos notar la diferencia entre un simplex de 3 vértices y un grafo con 3 nodos y 3 aristas. En el simplex la "cara" del triángulo es parte del mismo, por eso tiene dos dimensiones. El triángulo incluye tres simplices de una dimensión (sus lados) y tres simplices de cero dimensión (sus puntos). Podemos caracterizar un triángulo como el siguiente complejo simplicial: $\{A, B, C, \{A, B\}, \{B, C\}, \{C, A\}, \{A, B, C\}\}$ De la misma forma con 4 puntos formamos un simplex tridimensional que es un tetraedro, el tetraedro está formado por 4 simplices de 0 dimensiones (puntos) 6 simplices de 1 dimensión (segmentos), 4 simplices de dos dimensiones (triángulos) y un simplex de 3 dimensiones (el volumen del tetraedro).

Para transformar los datos en un complejo simplicial hay varias opciones, veremos las dos mas populares que son el complejo de Vietoris-Rips y el complejo de Cech.

18.2.1 Complejo Vietoris-Rips

El complejo de Vietoris-Rips se construye en base a un parámetro ϵ que es la distancia máxima para que dos puntos sean vecinos. Es decir que cada nodo queda conectado con todos los nodos que estén a distancia ϵ o menos. Esto es equivalente a construir esferas de radio ϵ alrededor de cada punto y luego tender una arista con todos los nodos que queden comprendidos dentro de esta esfera n-dimensional.

La figura 18.3 muestra una nube de puntos, las esferas de radio ϵ alrededor de los mismos y el complejo simplicial formado. Notemos que cuando 3 puntos

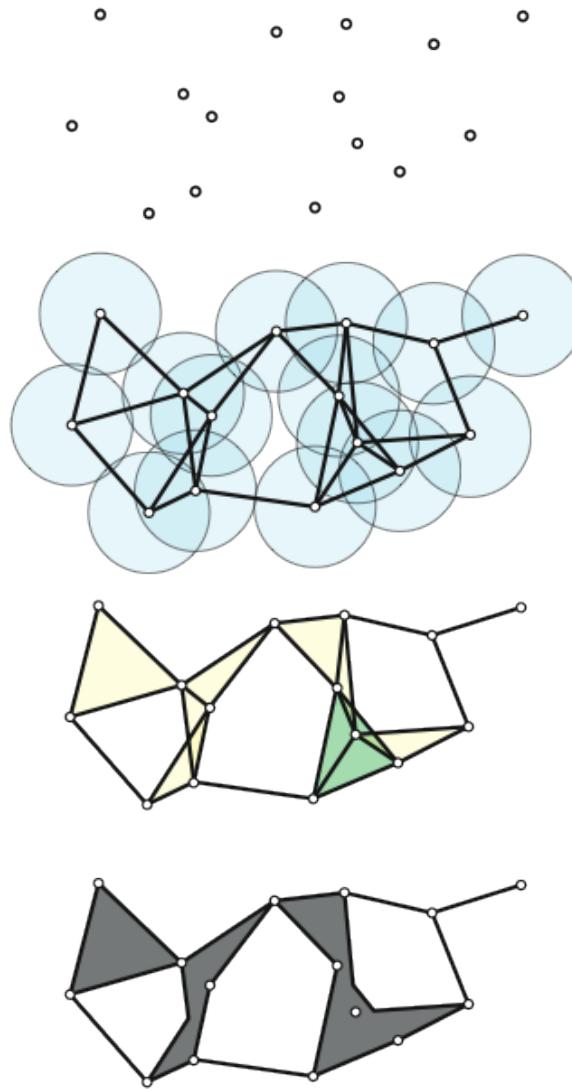
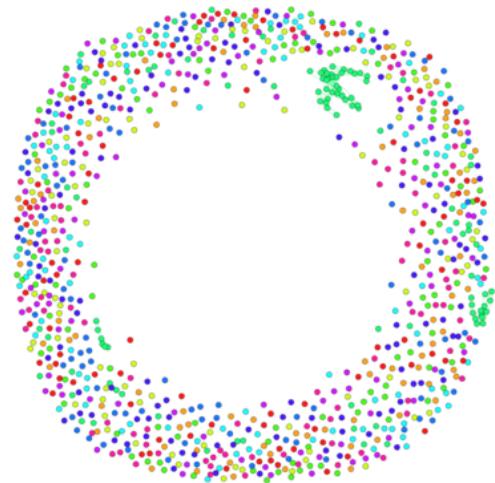


Figure 18.3: Complejo Vietoris Rips

están a distancia menor o igual a ϵ entre sí forman un triángulo cuya superficie coloreamos en amarillo, el volumen de los tetraedros lo coloreamos en verde.

A continuación mostramos el grafo resultante para MNIST y diferentes valores de ϵ

Como puede verse con diferentes valores de “ k ” o ϵ de se obtienen grafos completamente diferentes.

Figure 18.4: Complejo Vietoris-Rips con $\epsilon = 3$ Figure 18.5: Complejo Vietoris-Rips con $\epsilon = 10$

18.2.2 Complejo de Cech

El complejo de Cech es similar al de Vietoris-Rips pero la condición por la cual unimos los puntos de nuestro grafo es diferente.

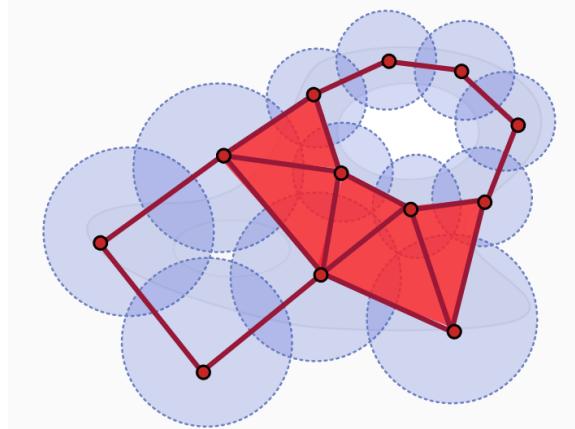


Figure 18.6: Complejo de Cech

En el complejo de Cech definimos un radio ϵ alrededor de cada punto y unimos dos puntos si las hiper-esferas de radio ϵ alrededor de los puntos se intersectan. Notar que esta condición es diferente a la del complejo Vietoris-Rips.

En general puede usarse uno u otro complejo para la construcción de grafos a partir de los datos, como veremos mas adelante no nos interesa exactamente el grafo que se forma sino como evoluciona el grafo a medida que variamos ϵ .

18.3 Homología Persistente

Como hemos visto el complejo simplicial que podemos construir depende enormemente del valor elegido para ϵ , si estudiamos la topología de un complejo simplicial estaremos estudiando la topología en función de ϵ , sin embargo la topología de los datos debería depender exclusivamente de los datos y no de un hiperparámetro externo. La solución es estudiar la topología de los datos para *todos* los valores de ϵ es decir que iremos variando ϵ y estudiando que complejo simplicial se forma.

La figura 18.7 nos muestra la formación de un complejo simplicial Vietoris-Rips a medida que variamos ϵ para un conjunto de datos que originalmente forma un anillo. Como podemos ver cuando ϵ es muy pequeño tenemos varios componentes disconexos que se van agregando entre sí, en algún punto en el medio aparece la topología de anillo y finalmente si usamos un ϵ muy grande nos quedando simplices de mas y mas dimensiones hasta que eventualmente

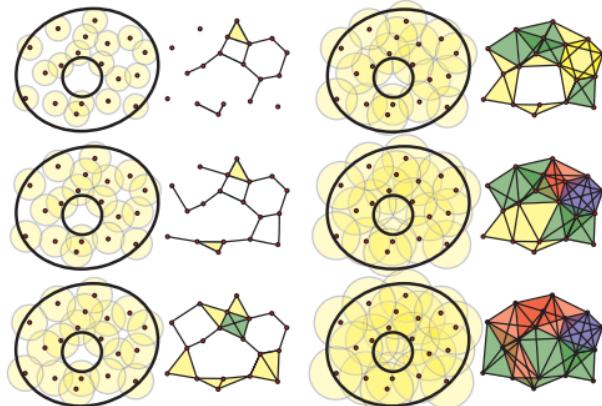


Figure 18.7: Homología Persistente

un cierto ϵ va a abarcar todos los puntos y nos quedará un único símplice gigante de tantas dimensiones como puntos.

El concepto clave detrás de la Homología Persistente es que el complejo que realmente representa la topología de los datos es el que mayor cantidad de tiempo persiste a medida que variamos ϵ . Esto lo podemos detectar graficando los llamados *números de Betti* en un diagrama que llamamos *barcode*

18.3.1 Números de Betti y Barcodes

Los números de Betti (β_i) miden la cantidad de "agujeros" i -dimensionales en un complejo simplicial. Resulta ser que los "agujeros" son la clave para estudiar la topología de los objetos ya que son los que hacen a la forma de los mismos.

El número de Betti cero (β_0) mide la cantidad de agujeros de 0 dimensiones, es decir la cantidad de componentes conexas en un complejo simplicial. El número Betti 1 (β_1) mide la cantidad de agujeros de 1 dimensión (loops), un círculo tiene uno. β_2 mide la cantidad de "agujeros" de dos dimensiones por ejemplo el interior de una esfera.

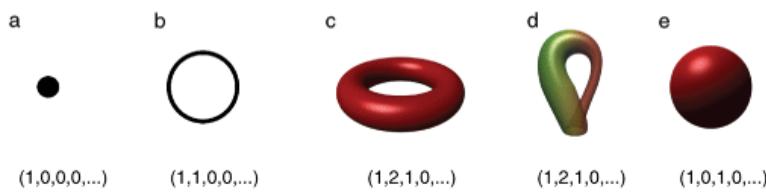


Figure 18.8: Números Betti

La figura 18.8 nos muestra el valor de β_0 , β_1 y β_2 para diferentes objetos. El "torus" por ejemplo tiene un único componente conexo, dos loops (el del centro y uno adentro) y un agujero de dimension 2 que es el interior de la dona. La esfera tiene un componente conexo, ningún loop y un "agujero" bidimensional que es su interior.

18.3.2 BarCodes

Un Barcode es una representación del valor de los diferentes números de Betti en función de ϵ en el eje x pondremos a ϵ y en el eje y los diferentes valores que puede tomar β_i , por ejemplo 0,1,2,3. Luego si para un cierto ϵ β_i toma el valor y marcamos un punto en su lugar, el resultado es una serie de "barras" que nos muestran la persistencia de los diferentes valores para el número de Betti a lo largo del tiempo. La idea es identificar los valores mas persistentes.

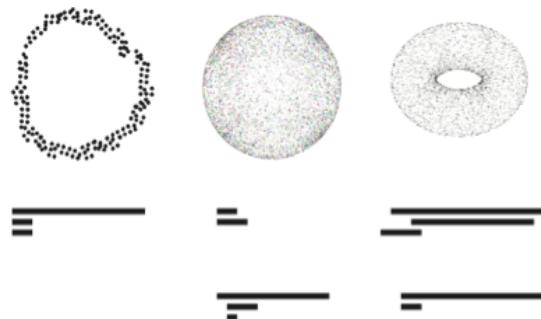


Figure 18.9: Barcodes

En la figura 18.9 vemos los barcodes para β_1 y β_2 para diferentes figuras. Podemos ver que para el anillo persiste una única barra para β_1 es decir que $\beta_1 = 1$ y eso es lógico porque el anillo tiene un solo loop. Para la esfera ninguna de las barras persiste por lo tanto $\beta_1 = 0$ y para el torus vemos dos barras por lo que $\beta_1 = 2$. De la misma forma para β_2 vemos que el valor es 1 para la esfera y el torus y cero para el anillo.

Los barcodes pueden ofrecer información a veces un tanto confusa pero esto no es mas que el reflejo de los datos mismos.

La figura 18.10 nos muestra un set de datos complejo y los barcodes para β_0 , β_1 y β_2 llamados H_0 , H_1 y H_2 en este plot. Podemos ver que es claro que hay un solo componente conexo y que $\beta_2 = 0$ pero el valor de β_1 es confuso, la cantidad de loops en este set de datos podría ser 1,2,3 o 4. Observando el set de datos vemos que es muy irregular y que efectivamente podríamos interpretar que existe un solo loop central, dos o tres o incluso cuatro. Tal vez la opción de tres loops sea la mas aceptable por consenso en este set de datos en particular.

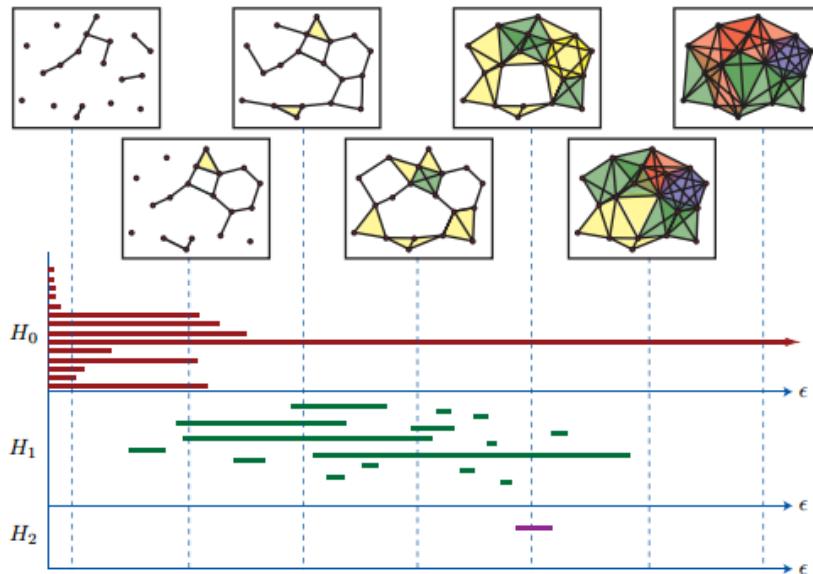


Figure 18.10: Barcodes para un set de datos complejo

18.3.3 Otros puntos de Análisis

El estudio de los barcodes de diferentes sets de datos nos da una idea de la topología del set de datos y nos permite comparar sets de datos entre sí, sabemos que sets de datos que tienen igual topología desde el punto de vista topológico en realidad son la misma cosa (!) [Carlson].

Veamos por ejemplo la cantidad de componentes conexos (β_0) para MNIST en contraste con un set de datos al azar. Notar que ploteamos el valor de β_0 y esto no es un barcode.

El plot nos muestra que MNIST no se comporta como un set de datos aleatorio, es decir que hay una cierta estructura en los datos que hacen que los clusters que se forman con diferentes valores de epsilon sean muy diferentes a los que se formarían con un grafo aleatorio. En concreto llegamos a tener un grafo conexo (clusters=1) con un epsilon bastante mayor al necesario para un grafo aleatorio. Esto permite deducir la existencia dentro de MNIST de subgrafos fuertemente localizados en donde sus miembros están muy cercanos unos a otros pero a su vez cada uno de estos subgrafos está alejado de los otros subgrafos. Es decir que el gráfico nos permite deducir la existencia de clases en MNIST, en otras palabras que hay diferentes grupos o clusters de datos, en nuestro caso, por supuesto, dígitos.

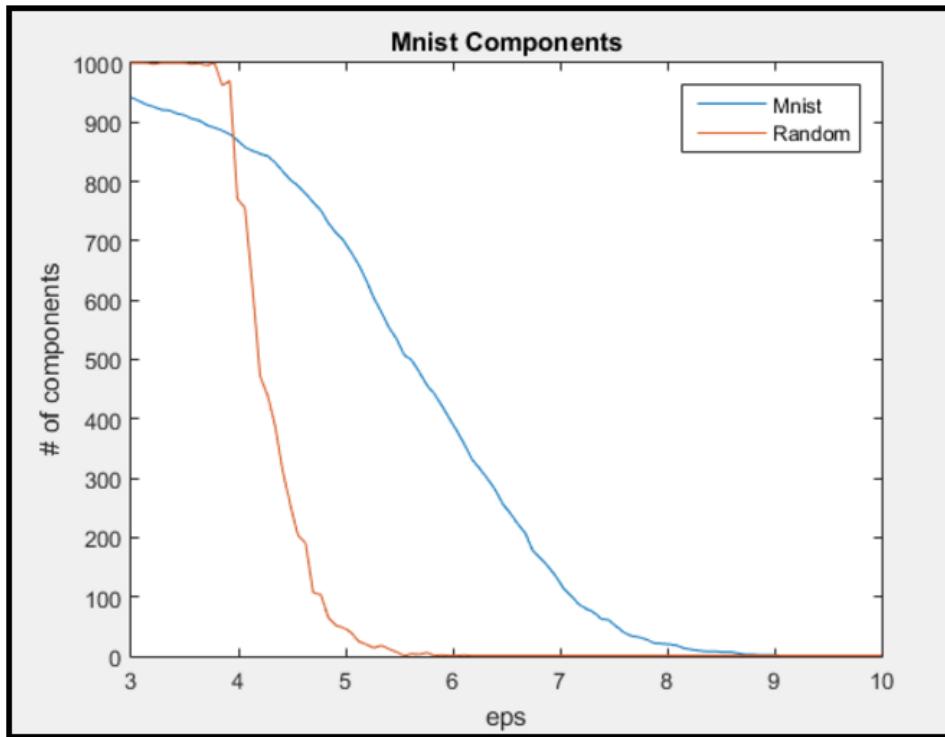


Figure 18.11: Cantidad de Components en MNIST vs Datos Aletorios

18.4 Algoritmo Mapper

Mapper [Singh-Memoli-Carlson] es un algoritmo para el análisis topológico de los datos, se basa en la creación de un grafo, similar a los que hemos visto pero con características diferentes. El algoritmo cuenta con varios pasos el primer paso es aplicar un filtro a los datos de forma tal que cada dato quede representado por un número real. Se puede usar cualquier tipo de filtro siempre y cuando cada dato quede representado por un número. A estos filtros se los llama “lentes” ya que, haciendo una analogía, equivalen a cambiar de lente en una cámara para ver las cosas de formas diferentes.

18.4.1 Lentes Topológicos Comunes

Entre los filtros mas comunes podemos mencionar los siguientes:

- i-ésima dimensión
- proyección aleatoria: (proyectamos cada punto mediante un mismo vector aleatorio)

- eccentricidad: promedio de las distancias entre el punto y todos los demás
- centralidad: distancia máxima desde el punto a cualquier otro punto del set de datos
- distancia al k-vecino mas cercano: distancia entre el punto y el k-ésimo vecino. ($k > 1$ de lo contrario vale 0 para todos los puntos).
- distance to measure: Promedio de las distancias entre el punto y sus k-vecinos mas cercanos (si $k = m$ entonces es eccentricidad)
- n-ésimo autovector de la SVD (un filtro muy popular)

Así como hemos definido estos filtros/lentes se pueden definir docenas o incluso cientos más, el único requisito es obtener un vector de “m” valores reales correspondientes a los “m” puntos de nuestro set de datos.

A continuación mostramos el resultado de algunos filtros sobre un set de datos muy simple representando una dona.

Como podemos ver diferentes lentes generan un mapeo de los datos completamente diferente por lo que es muy importante probar este algoritmo con diferentes lentes para ver cual nos da los resultados que son más lógicos y sencillos de interpretar.

Una vez que tenemos los datos filtrados por un cierto lente el siguiente paso es cubrir los datos con un cierto número de grupos con solapamiento en base al valor del filtro aplicado.

18.4.2 Cubriendo los Datos

Una vez que aplicamos el filtro tenemos a nuestros datos representados por números reales. El siguiente paso consiste en particionar los datos en “r” grupos con solapamiento.

A “r” lo vamos a denominar “resolución” y al nivel de solapamiento “s” siendo el mismo un número entre 0 y 1. Si usamos, por ejemplo, $r=20$ y $s=0.2$ estamos indicando que queremos particionar a nuestros datos en 20 grupos con un 20% de solapamiento entre los mismos.

Por ejemplo supongamos que tenemos los siguientes valores para un conjunto de 14 (catorce) datos:

1, 3, 5, 8, 11, 12, 13, 21, 23, 24, 28, 31, 32, 40

El intervalo total que tenemos es el [1,40] ahora queremos subdividir este intervalo en una cierta cantidad de grupos con solapamiento.

Con 10 intervalos y 25% de solapamiento el segmento es de longitud 5.0323. Es decir que nuestro primer grupo es [1,6.0323] el segundo grupo empieza en $1 + (5.0323 * (1 - 0.25))$ es decir 4.7742 entonces el segundo intervalo es: [4.7742, 9.8065] los diez intervalos son entonces:

```
[1,6.0323]
[4.7742,9.8065]
[8.5485,13.5808]
```

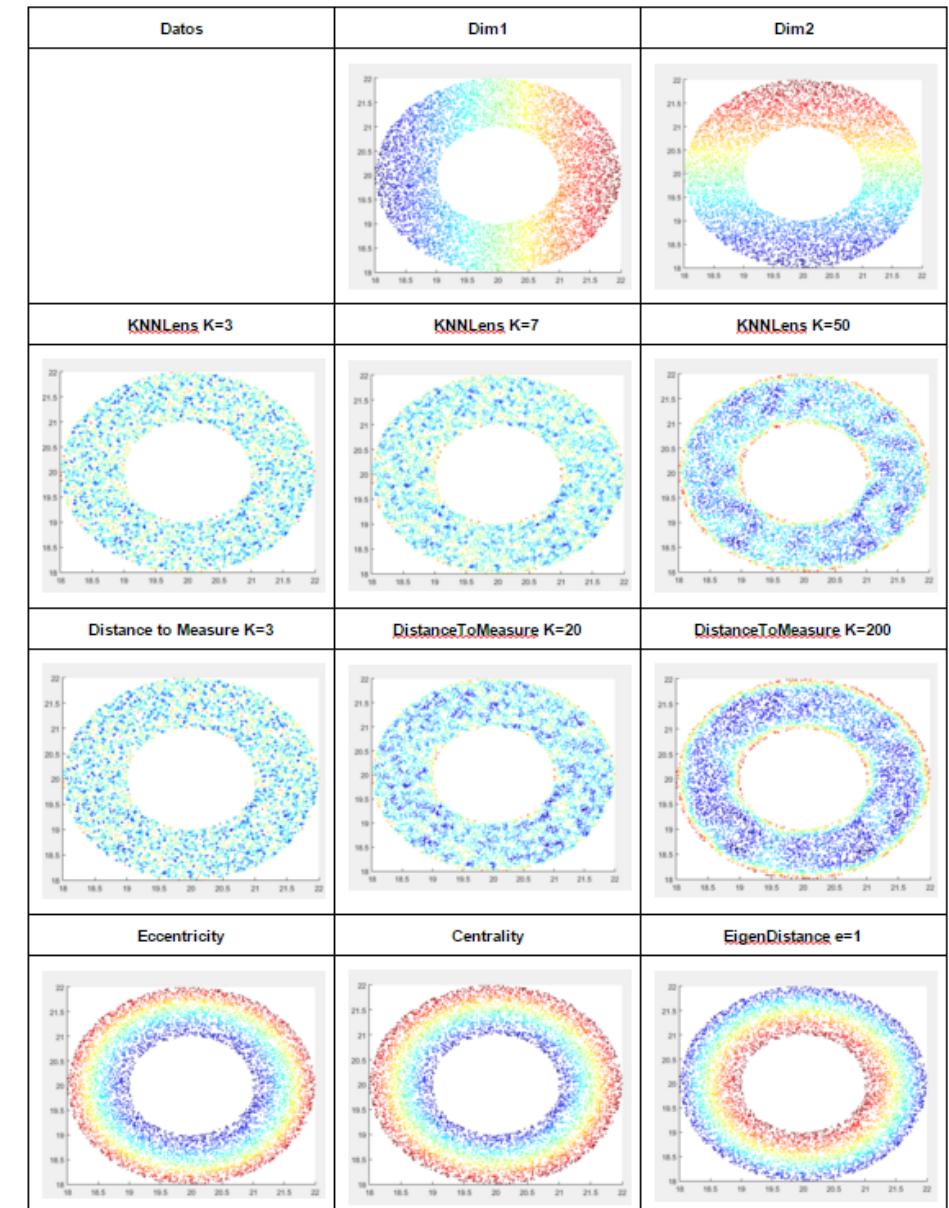


Figure 18.12: Lentes topológicos

[12.3227, 17.3550]
 [16.0969, 21.1292]
 [19.8711, 24.9034]

```
[23.6454,28.6777]
[27.4196,32.4519]
[31.1938,36.2261]
[34.9680,40]
```

En base a estos intervalos los elementos de nuestro set de datos quedan repartidos de la siguiente forma:

```
[1,6.0323] {1,3,5}
[4.7742,9.8065] {5,8}
[8.5485,13.5808] {11,12,13}
[12.3227,17.3550] {13}
[16.0969,21.1292] {21}
[19.8711,24.9034] {21, 23,24}
[23.6454,28.6777] {24,28}
[27.4196,32.4519] {28,31,32}
[31.1938,36.2261] {32}
[34.9680,40] {40}
```

Vamos que quedan definidos los diez grupos y que elementos de nuestros datos quedan asignados a cada grupo. Cada grupo es una partición de nuestros datos (con reemplazo).

La siguiente fase del algoritmo es aplicar clustering dentro de cada grupo.

18.4.3 Fase de Clustering

En la fase de clustering vamos a aplicar dentro de cada grupo un algoritmo de clustering. Es necesario poder detectar de forma automática el número de clusters dentro de cada grupo ya que lo desconocemos. Puede usarse cualquier algoritmo de clustering y cualquier método para seleccionar la cantidad de clusters. En nuestro código hemos elegido clustering jerárquico como algoritmo de clustering. Para seleccionar la cantidad de clusters a crear hemos elegido el siguiente método:

- En cada paso clustering jerárquico une dos clusters y nos devuelve la distancia entre los mismos.
- Es decir que tenemos como resultado un vector de distancias
- Estas distancias las discretizamos en una cierta cantidad de buckets (sin solapamiento)
- El primer bucket vacío nos da la cantidad de clusters a usar.

Este método ya lo vimos en detalle al estudiar clustering jerárquico.

El resultado es que por cada grupo de nuestros datos tenemos una cierta cantidad de clusters formados, desde 1 hasta la cantidad de puntos en el grupo. En conjunto el resultado final es una cantidad de clusters de puntos de nuestro set de datos. La última fase del algoritmo consiste en armar el grafo.

18.4.4 Construcción del Grafo

Para construir el grafo final vamos a tomar a cada cluster como un nodo y vamos a conectar entre sí a con aristas a los clusters que tengan puntos en común. Esta etapa es relativamente sencilla.

18.4.5 Ejemplo

Para nuestro set de datos “donut” aplicamos el filtro $\text{LensX}(\text{donut}, 1)$ que nos devuelve para cada punto la coordenada “x”. Y usamos como resolución 10 grupos con 50% de solapamiento. Usamos 10 buckets para discretizar los resultados de clustering jerárquico.

Este es el set de datos y los puntos coloreados de acuerdo al valor del lente:

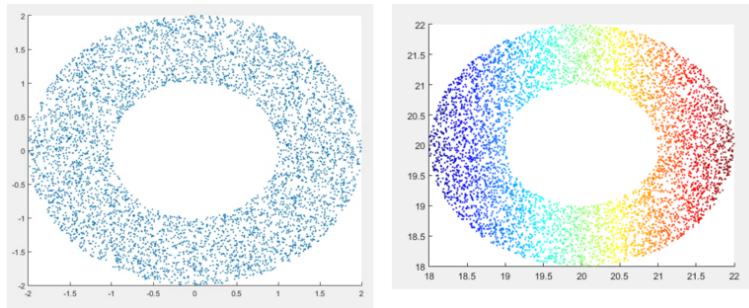


Figure 18.13: Ejemplo Mapper

Y el resultado luego de aplicar el algoritmo mapper con los parámetros indicados:

Los nodos fueron coloreados de acuerdo al promedio del valor del lente para los puntos dentro del cluster. Es sencillo ver como el algoritmo nos ha creado una representación topológica de nuestros datos, tenemos los “bordes” de la dona representados como dos “flares” y el “loop” representado como un ciclo dentro del grafo. Notemos que esta representación es invariante a transformaciones geométricas sobre la dona, si la estiramos o aplastamos un poco el resultado de Mapper será prácticamente el mismo.

Los “flares” y los “loops” son las características topológicas que nos permiten entender los datos, es de destacar que estas estructuras no aparecen en sets de datos aleatorios. Por ejemplo podemos construir un set de datos aleatorio de igual cantidad de puntos que nuestra dona y aplicar mapper con los mismos parámetros.

El resultado es una cadena de clusters sin estructura, aunque la ausencia de estructura es una estructura debemos visualizar la diferencia entre un conjunto de puntos aleatorios y los puntos en nuestra “dona”.

Es importante destacar la diferencia entre el análisis topológico y clustering.

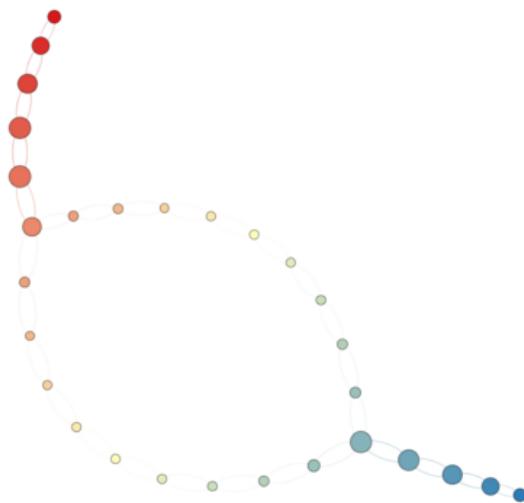


Figure 18.14: Resultado de Mapper

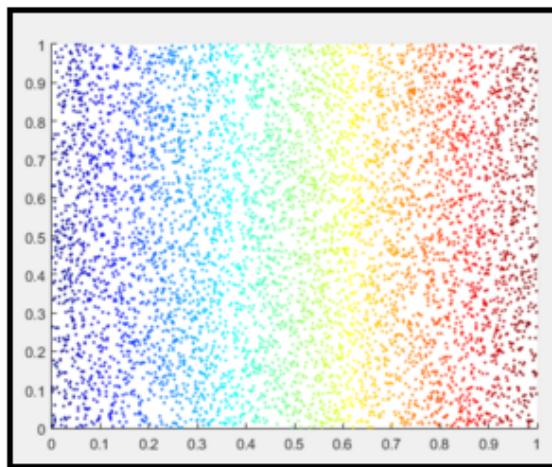


Figure 18.15: Set de Datos Aleatorio

La figura 18.17 muestra a la izquierda un set de datos con 4 clusters identificables, a la derecha mostramos la representación topológica de los datos usando Mapper. Observemos que un algoritmo de clustering es capaz de detectar los clusters igual que como lo hace el análisis topológico pero lo que no puede capturar es la conexión que existe entre estos clusters, es decir de que forma

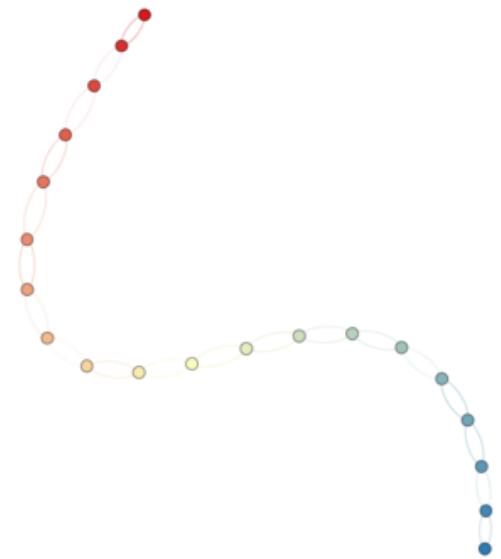


Figure 18.16: Set de Datos Aleatorio Resultado de Mapper

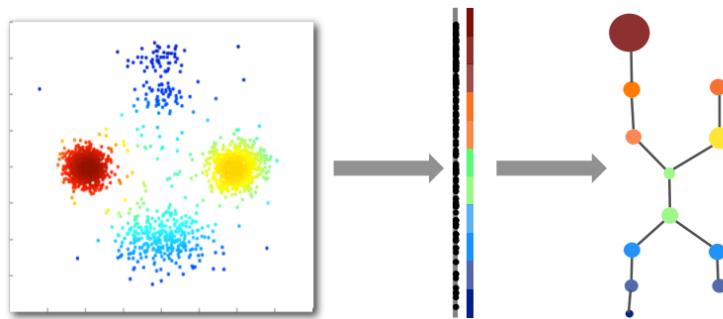


Figure 18.17: TDA vs Clustering

se produce la transición entre puntos que están dentro de un cluster hacia los otros clusters, esto es parte de la información contenida en los datos y un simple algoritmo de clustering la pierde(!).

Veamos un ejemplo bastante divertido. Tenemos una nube de puntos que representan un elefante en 3D.

Mostramos el set de datos y coloreamos en función del valor de la tercera dimensión.

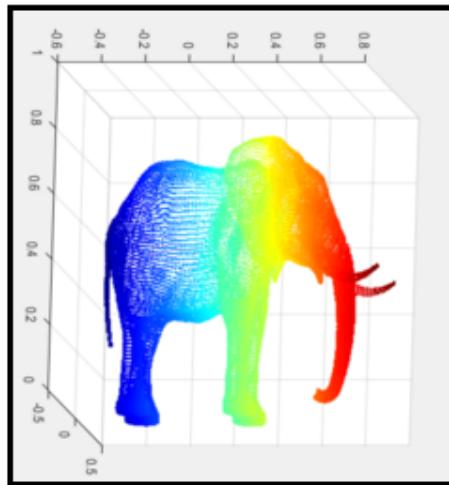


Figure 18.18: Set de Datos Elefante

Aplicamos mapper con resolución=50 y solapamiento 0.5 el resultado es el siguiente:

Lo que vemos es una representación topológica de un elefante. En rojo aparece la trompa con los colmillos, luego en naranja las dos orejas y la cabeza. Las patas delanteras (amarillo), luego el cuerpo (verde) y las patas traseras y la cola. Usando diferentes valores para la resolución y solapamiento podemos obtener diferentes representaciones.

18.4.6 Ejemplo

Analizaremos ahora un set de datos usando el algoritmo mapper para mostrar el proceso completo de análisis topológico usando este algoritmo.

Nuestro set de datos consiste de una encuesta de 11 preguntas realizadas a 20.000 personas en USA, las preguntas indican el grado de aceptación de cada persona en 11 categorías diferentes:

- churches
- armed forces
- the press
- television
- labor unions
- the police

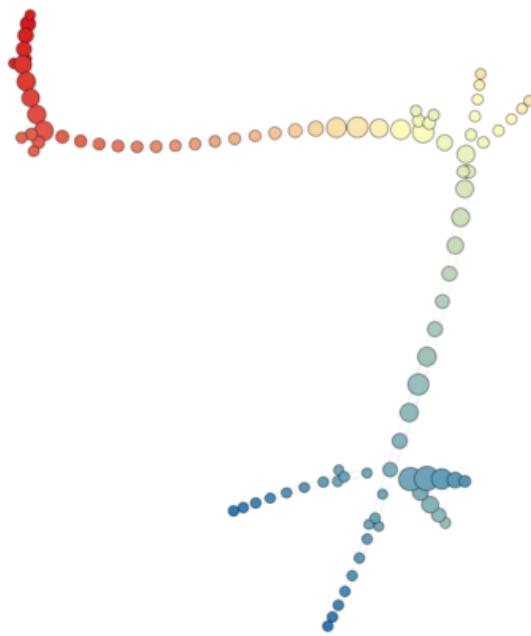


Figure 18.19: Set de Datos Elefante Resultado de Mapper

- the courts
- federal government
- political parties
- the civil service
- universities

Nuestro set de datos es entonces una matriz de 20000x11 en donde cada celda es un número de 1 a 10 en donde 10 indica un alto nivel de confianza y un 1 indica la ausencia total de confianza en la institución.

Para el análisis se usará la distancia de Mahalanobis (Distancia Euclídea Normalizada por Varianza) y como lentes topológicos usaremos el primer y segundo vector de la base de la SVD. Estos son los vectores que capturan la mayoría de la varianza en el set de datos como vimos al estudiar la SVD.

Luego de aplicar Mapper tenemos un primer vistazo de nuestros datos:

Inmediatamente podemos ver que Mapper nos da información muy interesante acerca de nuestros datos, podemos distinguir 4 grupos o clusters, unidos entre sí, uno en la izquierda (grande), uno en la parte superior, uno a la derecha

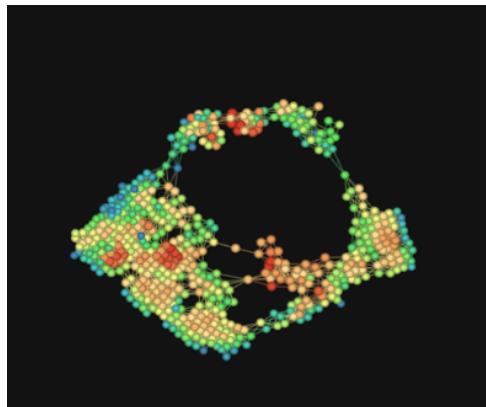


Figure 18.20: Resultado de Mapper

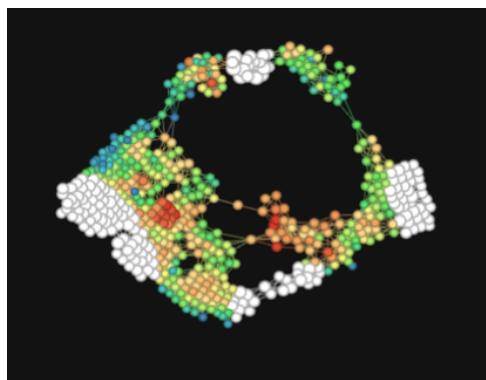


Figure 18.21: Identificando 4 clusters

y un pequeño grupo en la parte inferior. Podemos ahora consultar cuáles son los datos dentro de cada uno de estos grupos para ver cuáles son sus características.

Si analizamos cada uno de estos grupos descubrimos cuáles son los factores que tienen en común los miembros de cada grupo:

El grupo "B" que es el de la izquierda, el mas numeroso, tiene un alto grado de confianza en la mayoría de las instituciones, siendo la confianza un poco más débil en la policía, iglesia y fuerzas armadas.

El grupo "A" tiene muy poca confianza en cualquier institución.

El grupo "C" tiene un alto nivel de confianza en la policía, fuerzas armadas y la iglesia y un grado menor de confianza en el resto de las instituciones.

El grupo "D" se caracteriza por un bajo nivel de confianza en la mayoría de las instituciones pero un alto nivel de confianza en la iglesia.

De este análisis y la visualización de nuestros datos ya podemos sacar todo

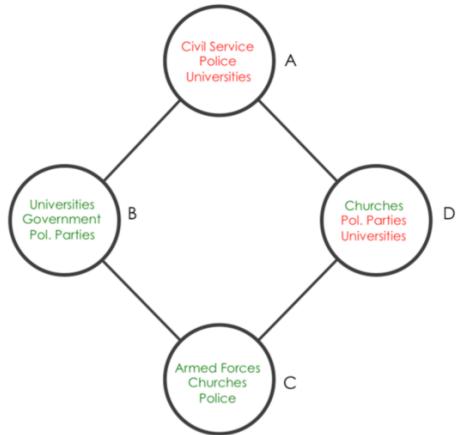


Figure 18.22: Características de cada Cluster

tipo de conclusiones muy interesantes tanto políticas como sociológicas y usarlas para las aplicaciones que se nos ocurran.

Observemos que existe un pequeño grupo flotante dentro de los demás que podemos llamar grupo "E".

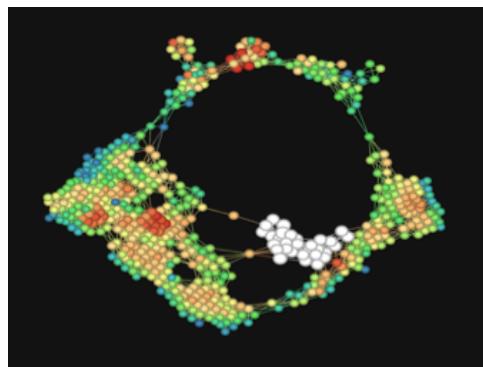


Figure 18.23: Grupo E

El grupo "E" tiene un menor nivel de confianza en la iglesia, policía y fuerzas armadas pero también un menor nivel de confianza en los bancos y partidos políticos. Podríamos decir que se trata de un grupo menos autoritario que el "C" pero mas cínico que el "B" y mucho menos creyente que el "D".

Si acompañamos este tipo de encuesta de una encuesta política de intención de voto podemos darnos cuenta cuales son los grupos que apoyan a uno u otro candidato o lo que es mas valioso cuáles son los grupos que podrían cambiar de

opinión de forma tal de poder concentrar el foco de una campaña en ellos.

18.5 Conclusiones

Mapper nos permite obtener una representación topológica de nuestros datos que luego podemos usar para detectar patrones que ocurren en los datos, entender la forma de los mismos y poder en base a observaciones detectar cosas que son muy difíciles de observar sobre los datos en crudo.

Mediante el uso de diferentes filtros topológicos y parámetros podemos obtener diferentes visualizaciones de nuestros datos, cada una de estas visualizaciones puede mostrar loops, flares y otras formaciones interesantes, el objetivo del análisis topológico es analizar cuáles son los datos que aparecen en dichas estructuras y por qué. Estas herramientas se han aplicado con éxito al estudio del padrón electoral, la detección de fraudes en transacciones financieras, el estudio genético de diferentes patologías en el área de salud e incluso en el deporte analizando de qué forma podemos clasificar o agrupar a los jugadores de un determinado deporte en base a sus características topológicas, descubriendo posiciones o categorías que no sabíamos que existían pero que se ponen de manifiesto mediante los datos.

Chapter 19

Metadatos

The web as I envisaged it, we have not seen it yet. The future is still so much bigger than the past.

- Tim Berners Lee

Este capítulo es sobre abstracciones, en concreto vamos a embarcarnos en la enorme tarea de representar cualquier tipo de información, y no solo vamos a intentar desarrollar un modelo que nos permita representar cualquier cosa sino que el modelo va a ser simplemente un conjunto de triples. Antes de llegar a estas conclusiones vamos a empezar a hablar de *metadatos* que son los datos que describen a los datos.

El mundo está lleno de metadatos y trabajamos con ellos todos los días sin darnos cuenta. Si tomamos, por ejemplo, un DVD los datos estarán representados mediante un archivo de video de algún formato pero además de esto el DVD contiene todo tipo de metadatos. El título de la película es un metadato, la fecha en que fue realizada, los actores que participan, el director, la duración de la misma, etc. Los metadatos nos sirven para entender los datos, para saber de qué trata el archivo de video que tenemos en el DVD, cuánto dura, quién lo hizo, en qué fecha, etc etc.

Los metadatos sirven para que las computadoras puedan entender los datos, por ejemplo una computadora que analiza un archivo de video puede no entender jamás que se trata de "Pulp Fiction", sin embargo con un simple metadato "title" podemos darle esa información. Con esa información la computadora puede entender que archivos de audio que dicen pertenecer a la película "Pulp Fiction" probablemente estén relacionados con el archivo de video anterior. Las computadoras necesitan de los metadatos porque los datos son muy difíciles de entender sin una estructura auxiliar que los describa.

Si nos enfocamos en la tarea de proveer de metadatos a las computadoras para que puedan entender los datos nos daremos cuenta que un problema significativo es que nombre debemos darle a los metadatos y qué significan los mismos. Por ejemplo el metadato "creator" para un DVD podría ser la persona

que grabó el DVD o tal vez el productor de la película, o el estudio que la lanzó o incluso el director de la misma. Un mismo metadato podría tener significados completamente diferentes. De la misma forma un mismo significado puede representarse con diferentes nombres, si nos referimos al autor de un libro el mismo podría llamarse "writer", "creator", o "author" en sistemas diferentes.

19.1 Dublin Core

Es fundamental, entonces, estandarizar tanto los nombres como los significados de los metadatos, una de las iniciativas mas populares al respecto es la llamada "Dublin Core" en donde Dublin no es la capital de Irlanda sino una ciudad del estado de Ohio. El Dublin Core (DC) define un conjunto de 15 metadatos que pueden usarse de forma genérica para cualquier tipo de dato. Por cada uno de estos metadatos el DC define cuál es su significado y como debe usarse el mismo. Los elementos del Dublin Core son los siguientes:

1. Title
2. Creator
3. Subject
4. Description
5. Publisher
6. Contributor
7. Date
8. Type
9. Format
10. Identifier
11. Source
12. Language
13. Relation
14. Coverage
15. Rights

Todos los elementos de Dublin Core son opcionales, se pueden repetir y pueden aparecer en cualquier orden.

Los elementos de Dublin Core son independientes de cualquier sintaxis, por ejemplo se pueden expresar en XML, en HTML, en texto plano, mediante grafos o cualquier otra sintaxis, un ejemplo de uso de elementos de Dublin Core en HTML es el siguiente:

```
<meta name="DC.Format" content="video/mpeg; 10 minutes">
<meta name="DC.Language" content="en" >
<meta name="DC.Title" content="La Historia del Dublin Core" >
```

Así como existe DC existen otras agrupaciones que definen metadatos standard para su dominios específicos, como por ejemplo los museos, las bibliotecas y los archivos. La definición de un set de metadatos permite que instituciones diferentes intercambien o consulten información sin necesidad de realizar una traducción entre el vocabulario de metadatos de una y la otra.

Dublin Code es un conjunto de propiedades, es decir cosas que podemos querer describir a partir de un dato, no es un standard para metadatos sino que es compatible con cualquier standard de metadatos que quiera usar estas propiedades. El standard de metadatos universalmente aceptado se llama RDF (Resource Description Framework)

19.2 RDF

RDF abarca un conjunto de estándares creados por la W3C que originalmente tenían como objetivo la representación de un modelo de metadatos y ha terminado extendiéndose a un método general para describir o modelar información. Informalmente podríamos decir que en algún momento alguien se dio cuenta que RDF en realidad sirve para expresar no solo metadatos sino cualquier cosa.

19.2.1 El Modelo RDF

El modelo de RDF es muy simple y se basa en pocos elementos. Un modelo RDF es una colección de *statements* en donde cada statement es una tripla de tipo (*subject, predicate, object*). El "sujeto" (*subject*) es lo que queremos describir también llamado *resource* (recurso). En RDF cualquier cosa que se pueda describir es un recurso. El predicado es una propiedad o atributo del sujeto. El objeto finalmente es el valor que tiene dicha propiedad.

Por ejemplo si decimos ("Mona Lisa", "DC:Creator", "Leonardo Da Vinci") estamos describiendo el sujeto "Mona Lisa", la pintura y de este recurso estamos dando la propiedad "DC:Creator" en donde DC refiere a un elemento del Dublin Core y el valor de la propiedad es "Leonardo Da Vinci". RDF tiene pocas reglas pero son fundamentales para poder entender el modelo.

1. El sujeto siempre es un recurso y puede ser anónimo o identificarse con una URI
2. El objeto puede ser otro recurso o un literal

Todos los recursos en RDF se identifican mediante una URI (uniform resource identifier). Una URL es un caso particular de URL en donde el protocolo es http. Es muy común usar URLs como identificadores de recursos en RDF, por ejemplo "http://www.art.org/paintings/MonaLisa". La URL no tiene porque

existir físicamente, es decir no tiene que ser algo alcanzable por un browser, es simplemente un texto, una clave o identificador.

RDF es completamente independiente de la sintaxis, es un modelo para representar información y no una sintaxis. Existen muchas sintaxis posibles para RDF, de todas estas veremos los grafos y Turtle.

El modelo de RDF basado en triples puede representarse mediante grafos dirigidos en donde usaremos óvalos para representar recursos, aristas dirigidas para representar las propiedades y rectángulos para representar literales.

19.2.2 RDF mediante Grafos

Supongamos que queremos representar "La capital de Francia es París fundada en el año 300 b.c."

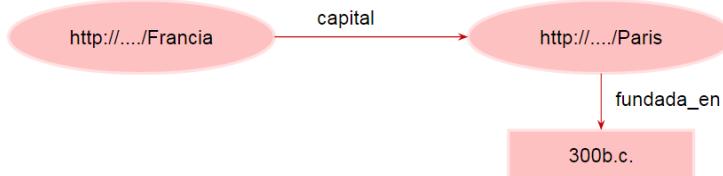


Figure 19.1: RDF primer ejemplo

La figura 19.1 nos muestra la representación mediante grafos de nuestra frase modelada como RDF. Tenemos dos recursos: Francia y París y un literal que es el año en que fue fundada París.

En general todo aquello que puede describirse debe representarse como un recurso y es un error representarlo mediante un literal, aun si no nos interesa dar ninguna propiedad de dicho recurso. Los únicos literales son números y strings con el cuidado de distinguir el string que es el nombre/label de un Recurso del Recurso en si mismo.

19.2.3 Recursos Anónimos

En algunos casos podemos contar con recursos que no nos interesa identificar, por ejemplo la dirección de un lugar o donde vive una persona es un recurso pero no nos interesa que tenga un identificador propio. ¿Por qué? porque es improbable que queramos usar la dirección como un sujeto de nuestros datos, estamos interesados en las propiedades de dicha dirección y en cuál es el recurso cuya propiedad "dirección" tiene esos valores.

En el ejemplo vemos como el recurso "Pedro" tiene una propiedad "dirección" que es un recurso anónimo y que a su vez tiene las distintas propiedades que permiten especificar un lugar físico. En el ejemplo podemos argumentar que

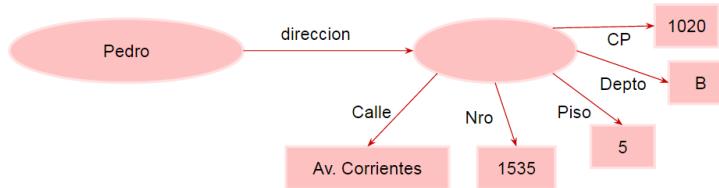


Figure 19.2: Recursos Anónimos

”Av Corrientes” no debería ser un literal sino un recurso ya que puede interesarnos como algo que queremos describir. Por ejemplo podemos hablar de su historia, extensión, barrios que atraviesa, etc etc.

19.3 Turtle

Turtle es una sintaxis para RDF que es legible tanto para programas como personas. Es nuestra sintaxis favorita para modelar usando RDF.

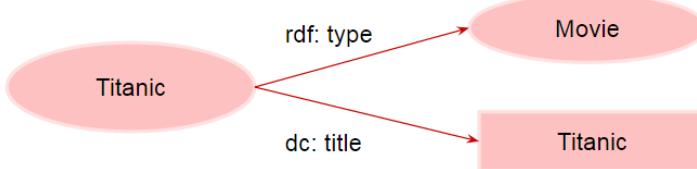


Figure 19.3: Ejemplo Turtle

La figura 19.3 nos muestra un sencillo modelo RDF en donde describimos la película "Titanic" y damos la propiedad "rdf:type" y "dc:title" (dc=Dublin Core). En Turtle la sintaxis es la siguiente:

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix mv: <http://example.org/mv/> .

mv:Titanic rdf:type mv:movie .
mv:Titanic dc:title \Titanic .

```

En el encabezado declaramos prefijos cómodos como "rdf", "dc", "mv" para las URIs o URLs que vamos a usar como identificadores, de esta forma mv:movie en realidad refiere a <http://example.org/mv/movie>.

Cada "statement" es una tripla de la forma sujeto predicado objeto y termina en un punto. La expresión "rdf:type" que indica a que "clase" pertenece un recurso puede reemplazarse simplemente por "a", es decir: "Titanic a mv:movie." es un statement válido en Turtle. Si queremos dar varias propiedades de un mismo objeto podemos hacerlo separándolas con ";".

```
mv:Titanic a mv:movie;
mv:director mv:JamesCameron;
mv:year 1994.
```

Los literales si son números pueden escribirse directamente y si son strings deben escribirse entre comillas dobles.

Podemos también dar varios valores para una misma propiedad usando "," para separarlos.

```
mv:Titanic mv:actor mv:LeonardoDiCaprio,
mv:KateWinslet.
```

Esto equivale a:

```
mv:Titanic mv:actor mv:LeonardoDiCaprio.
mv:Titanic mv:actor mv:KateWinslet.
```

Los recursos anónimos en Turtle se escriben entre corchetes:

```
ex:Pedro ex:address [ex:street ex:AvCorrientes;
ex:number 1535;
ex:city "CABA"].
```

19.4 Ontologías

Una ontología es una colección de información sobre un cierto dominio. Incluye la información en sí misma (datos) como así también las reglas que indican de que forma se representa la información en dicho dominio. Es un concepto mucho más amplio que una base de datos.

El lenguaje standard para expresar ontologías es OWL (Web Ontology Language) cuya sigla está mal escrita en honor al buho (owl) en Winnie Pooh que escribía mal su nombre. Un sub-set de OWL es RDF-Schema (RDFs) que es el que veremos en este capítulo ya que el tratamiento completo de OWL podría llevarnos todo un libro y a esta altura realmente no tenemos ganas de escribir otra vez la frase "Capítulo 1".

19.4.1 RDF-Schema (RDFs)

Es un subset de OWL y nos aporta un conjunto de clases y propiedades para crear ontologías, vamos a describir cuales son los diferentes elementos de RDFs y combinarlos con algunos elementos que ya conocíamos de RDF para armar el vocabulario completo que nos permitirá, representar todo el conocimiento de la humanidad (si el tiempo alcanza).

Clases

Los recursos en RDF pueden pertenecer a una o varias clases al mismo tiempo. Una clase es un recurso cuya clase es "rdf:Class", esto puede ser un tanto confuso! Para expresar pertenencia a una clase usamos "rdf:type" o en Turtle simplemente "a".

Por ejemplo si decimos "mv:Titanic a mv:movie." estamos expresando que Titanic pertenece a la clase "movie" implicitamente estamos diciendo algo de tipo "mv:movie a rdf:Class" pero esto no hace falta escribirlo ya que se deduce de lo anterior.

Algunas clases predefinidas son "rdfs:Class", "rdfs:resource", "rdfs:literal" o "rdfs:property" en general la clase se infiere a partir de los statements, si decimos por ejemplo "foaf:Juan foaf:age 20" inferimos que "foaf:age a rdfs:property" y que 20 es un literal pero todo eso no hay que escribirlo y en el caso de 20 ni siquiera podríamos escribirlo porque un literal no puede ser un sujeto, el hecho de que algo es un literal solo puede inferirse y nunca explicitarse.

Podemos crear una jerarquía de clases usando la propiedad "rdfs:subClassOf", por ejemplo: "mv:actor rdfs:subClassOf mv:person" lo que decimos allí es que todo actor es también una persona, esto quiere decir que todo lo que definamos para una persona aplica también a un actor y que cuando declaramos a un recurso como tipo "actor" automáticamente lo estamos declarando como tipo "persona" y por supuesto esto se infiere y no hay que escribirlo explicitamente.

Curiosamente podemos crear también una jerarquía de propiedades mediante "rdfs:subPropertyOf", por ejemplo "mv:actsIn rdfs:subPropertyOf mv:worksIn" aquí decimos que cualquier recurso que "actúa en" automáticamente también "trabaja en".

Domain y Range

La propiedad "rdfs:domain" se aplica a propiedades y nos indica cuáles son las clases que pueden ser sujeto de dicha propiedad. por ejemplo:

```
mv:age a rdf:property .
mv:age rdfs:domain mv:person .
mv:Juan a mv:person .
mv:Juan mv:age 23 .
```

Aquí decimos que "age" es un property y que "age" se aplica recursos de clase "person", luego declaramos a "Juan" como persona y decimos que tiene edad igual a 23. Notemos que la primera línea es redundante, se infiere de la segunda y no deberíamos haberla escrito pero lo hicimos por claridad. automáticamente al usar "domain" lo que está a la izquierda es de tipo "property" y lo que está a la derecha es de tipo "class" es decir una clase.

La propiedad "rdfs:range" define cuáles son los valores válidos de una propiedad. Por ejemplo:

```
mv:age rdfs:range rdfs:literal.
```

```
mv:actsIn rdfs:range mv:movie.
mv:actsIn rdfs:range mv:series.
```

Aquí decimos que la propiedad "age" solo puede tomar valores literales mientras que la propiedad "actsIn" puede tomar valores que sean de la clase "mv:movie" o "mv:series". Es importante destacar, una vez, más que en RDF una misma propiedad puede tomar n valores diferentes.

19.4.2 Otros elementos de RDFs

La propiedad "rdfs:label" se usa para escribir un identificador legible por un humano para un objeto. Por ejemplo "mv:Titanic rdfs:label "Titanic (1994)" en teoría todos los recursos deberían tener un label para que un sistema de consultas pueda mostrar los labels y no las URIs de los recursos que son mucho más crípticas.

La propiedad "rdfs:comment" sirve para hacer, precisamente, comentarios sobre los recursos.

La propiedad "rdfs:seeAlso" sirve para referenciar a otro recurso cuando accedemos a uno determinado. No es un alias sino que indica que es conveniente ver también la información del o los recursos referenciados.

19.5 SPARQL

Es un lenguaje de consultas para el modelo RDF, permite obtener información sobre los diferentes recursos almacenados en alguna base RDF. Su sintaxis está inspirada en SQL y un lenguaje de consultas que ya no existe más y se llamaba FLOWR.

La estructura de una consulta en SPARQL es:

```
# prefix declarations
PREFIX foo: <http://example.com/resources/>
...
# dataset definition
FROM ...
# result clause
SELECT [DISTINCT] ...
# query pattern
WHERE { ... }
# query modifiers
ORDER BY ...
LIMIT
```

En "PREFIX" declaramos los prefijos que vamos a usar igual que como hacíamos en Turtle.

"FROM" indica cuál será el lugar al cual vamos a consultar la información.

”SELECT” indica cuáles son las variables que vamos a mostrar como resultado de la consulta. Las variables en SPARQL se denotan de la forma ”?varname” es decir un signo de pregunta seguido de un nombre.

”WHERE” es donde escribimos la consulta propiamente dicha, es decir todas las condiciones que queremos cumplir para que un resultado aparezca como salida de la consulta.

Luego podemos escribir modificadores que afectan al resultado de la consulta como FILTER para filtrar ciertos resultados, ORDER BY para ordenarlos o LIMIT para limitar la cantidad de resultados que mostramos.

A modo de ejemplo vamos una consulta en donde queremos los nombres de los contactos de Tim Berners-Lee almacenados en: <http://dig.csail.mit.edu/2008/webdav/timbl/foaf.rdf>

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
FROM <http://dig.csail.mit.edu/2008/webdav/timbl/foaf.rdf>
SELECT ?name
WHERE {
    ?person foaf:name ?name .
}
```

Como podemos ver es muy simple, queremos ”?name” de forma tal que en nuestra base RDF exista una tripla que cumpla con ”?person foaf:name ?name” es decir listamos todas las personas que tienen nombre y de ellas mostramos el nombre.

Un ejemplo mas complejo: queremos todas las personas que son amigas entre sí.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
FROM <http://dig.csail.mit.edu/2008/webdav/timbl/foaf.rdf>
SELECT ?name1, ?name2
WHERE {
    ?person1 foaf:name ?name1.
    ?person2 foaf:name ?name2.
    ?person1 foaf:friend ?person2.
}
```

Es decir person1 tiene que tener nombre, person2 tiene que tener nombre y person1 tiene que ser amiga de person2, y listamos los dos nombres: name1 y name2. En el WHERE escribimos entonces cuales son las condiciones que tienen que cumplirse en nuestro modelo RDF es decir que triples tienen que existir para que un resultado sea válido, con este sencillo modelo y esta sintaxis simple podemos consultar cualquier modelo RDF y como hemos planteado que con RDF podemos almacenar cualquier conocimiento resulta ser que con SPARQL podemos averiguar cualquier cosa.

O al menos esa es la teoría....

Bibliography

- [Aggarwal,Hinneburg,Keim] Charu Aggarwal, Alexander Hinneburg, Daniel KEim. On the Surprising Behavior of Distance Metrics in High Dimensional Space
- [Achlioptas] Dimitris Achlioptas. Database friendly random projections Johnson-Lindestrauss with Binary Coins.
- [Alon-Matias-Szegedy] Noga Alon;Yossi Matias;Mario Szegedy. The space complexity of approximating the frequency moments.
- [Andoni] Alexandre Andoni; Piotr Indyk; Huy Nguyen; Ilya Razenshteyn. Beyond Locality Sensitive Hashing.
- [Andoni2] Alexandre Andoni; Piotr Indyk; Thijs Laarhoven; Ilya Razenshteyn. Practical and Optimal LSH for Angular Distance.
- [Attenberg] Joshua Attenberg;Kilian Weinberger. Collaborative Email Spam Filtering with the Hashing Trick.
- [Babenko-Lempitsky] Artem Babenko; Victor Lempitsky. The Inverted Multi Index.
- [Bahmani] Bahman Bahmani; Benjamin Mosley; Andrea Vattani; Ravi Kumar; Sergei Vasilevskii Scalable Kmeans plus plus.
- [Banko-Brill] Michele Banko, Eric Brill. (2001). Scaling to Very Very Large Corpora for Natural Language Disambiguation
- [Bawa] Mayank Bawa; Tyson Condie; Prasanna Ganesan. LSH Forest Self Tuning Indexes for Similarity Search.
- [Belazzougui-Botelho-Dietzfelbinger] Djamal Belazzougui, Fabiano Botelho, Martin Dietzfelbinger. Hash Displace and Compress
- [Belkin] Mikhail Belkin;Partha Niyogi. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation.
- [Beyer] Beyer K., Goldstein J., Ramakrishnan R., Shaft U. When is Nearest Neighbors Meaningful?
- [Botelho-Pagh-Ziviani] Fabiano Botelho, Rasmus Pagh, Nivio Ziviani. Simple and Space Efficient Minimal Perfect Hash Functions.
- [Bottou-Bengio] Leon Bottou;; Yoshua Bengio. Convergence Properties of the K-Means Algorithms
- [Burrows-Wheeler] M. Burrows; David J Wheeler. A block-sorting lossless data compression algorithm.

- [Campello-Moulavi-Sander] Ricardo J. G. B. Campello; Davoud Moulavi; Joerg Sander Density Based Clustering Based on Hierarchical Density Estimates.
- [Carlson] Gunnar Carlson. Topology and Data.
- [Charalampos] Charalampos;Tsourakakis. Fast Counting of Triangles in Large Real Networks.
- [Charikar] M Charikar. Similarity Estimation Techniques from Rounding Algorithms.
- [Chazelle] Bernard Chazelle;Nor Ailon. Approximate Nearest Neighbors and the Fast Johnson-Lindenstrauss Transform.
- [Chen-Guestrin] Tianqi Chen;Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. <https://drive.google.com/folderview?id=0B8rBD4QSqWnSRml5R3dsNEdGTHMusp=sharin+>
- [Clauset-Newman-Moore] Aaron Clauset; M. E. J. Newman; Cristopher Moore. Finding community structure in very large networks.
- [Clearly-Witten] Cleary, J.; Witten, I. Data Compression Using Adaptive Coding and Partial String Matching
- [Cormack-Horspool] Gordon Cormack;Nigel Horspool. Data Compression using Dynamic Markov Modelling.
- [Dasgupta] Sanjoy Dasgupta;Anupam Gupta. An elementary proof of the Johnson-Lindenstrauss Lemma.
- [Datar-Gionis-Indyk-Motwani] M. Datar; A Gionis; P Indyk; R. Motwani. Maintaining stream statistics over sliding windows.
- [Dean] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters
- [Dhillon] Inderjit S Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning
- [Dietzfelbinger] M. Dietzfelbinger; A. Karlin; K. Mehlhorn; F. Meyer auf der Heide; H. Rohnert; R. E. Tarjan. Dynamic perfect hashing upper and lower bounds.
- [Dimitropoulos] Xenofontas Dimitropoulos; Marc Stoecklin; Paul Hurley; Andreas Kind. The Eternal Sunshine of the Sketch Data Structure.
- [Dong] Wei Dong, Moses Charikar,Kai Li. (2011). Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures.
- [Eldridge-Belkin] Justin Eldridge; Mikhail Belkin. Beyond Hartigan Consistency Merge Distortion Metric for Hierarchical Clustering.
- [Flajolet-Martin] Philippe Flajolet; Nigel Martin. Probabilistic Counting Algorithms for Data Base Applications.
- [Flajolet] Philippe Flajolet;Eric Fusy; Olivier Gandouet; Frederic Meunier. The analysis of a near-optimal cardinality estimation algorithm.
- [Fredman-Komlos-Szemeredi] Michael Fredman; Janos Komlos; Endre Szemerédi. Storing a Sparse Table with O(1) Worst Case Access Time.
- [Good-Montjoye-Clauset] Good; Montjoye; Aaron Clauset. The performance of modularity maximization in practical contexts.
- [Gou] Janping Gou, Lan Du,Yuhong Zhang,Taisong Xiong. (2012). A New Distance-weighted k-nearest Neighbor Classifier

- [Grumwald] Peter Grumwald. A Tutorial Introduction to the Minimum Description Length Principle.
- [Guha-Rastogi-Shim] Guha, Sudipto; Rastogi, Rajeev; Shim, Kyuseok. CURE An Efficient Clustering Algorithm for Large Databases.
- [Hajebi] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, Hong Zhang. Fast Approximate Nearest-Neighbor Search with k-Nearest Neighbor Graph
- [Houle] Michael E. Houle,Hans-Peter Kriegel,Peer Kroger,Erich Schubert,Arthur Zimek. Can Shared-Neighbor Distances Defeat the Curse of Dimensionality?
- [Indyk-Motwani] Piotr Indyk; Rajeev Motwani; Aristides Gionis. Similarity Search in High Dimensions via Hashing.
- [Indyk-Andoni] Piotr Indyk; Alexandr Andoni. Near Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions.
- [Kalantidis-Avrithis] Yannis Kalantidis;Yannis Avrithis Locally Optimized Product Quantization for Approximate Nearest Neighbor Search.
- [Kang-Meeder-Faloutsos] U Kang;Brendan Meeder;Christos Faloutsos. Spectral Analysis for Billion Scale Graphs Discoveries and Implementation.
- [Kennedy] Christopher Kennedy; Rachel Ward. Fast Cross Polytope LSH.
- [Kleinberg] Jon Kleinberg. Authoritative sources in a hyperlinked environment.
- [Knoll-de Freitas] Byron Knoll;Nando de Freitas. A Machine Learning Perspective on Predictive Coding with PAQ.
- [Koren] Yehuda Koren. Factorization Meets the Neighborhood.
- [Lempel-Ziv-C] Lempel, A.; Ziv, J. On the complexity of finite sequences.
- [Leskovec] J Yang; Jure Leskovec. Overlapping Community Detection at Scale A Nonnegative Matrix Factorization Approach.
- [Louvain] Vincent D Blondel; Jean-Loup Guillaume; Renaud Lambiotte; Etienne Lefebvre. Fast unfolding of communities in large networks.
- [Muthukrishnan-Cormode] S. Muthukrishnan; Graham Cormode. The Count-Min Sketch and its Applications.
- [Page-Brin] Larry Page; Sergey Brin. The PageRank Citation Ranking Bringing Order to the Web.
- [Park-Leckie] Laurence A F Park; Christopher A Leckie; Kotagiri Ramamohanarao;James C Bezdek. Adapting spectral coclustering to documents and terms using latent semantic analysis.
- [Rendle] Steffen Rendle. Factorization Machines.
- [Rendle2] Steffen Rendle. Factorization Machines with LibFM.
- [Rissanen] Rissanen, J. Modeling by the shortest data description.
- [Rosvall-Bergstrom] Martin Rosvall; Carl T Bergstrom. Maps of random walks on complex networks reveal community structure.
- [Pagh] Rasmus Pagh Hash and Displace Efficient Evaluation of Minimal Perfect Hash Functions.

- [Pagh-Rodler] Rasmus Pagh; Flemming Rodler. Cuckoo Hashing.
- [Shannon] Shannon Claude. A Mathematical Theory of Communication
- [Shi] Qinfeng Shi; James Petterson; John Langford; Alex Smola; Alex Strehl; Gideon Dror; Vishy Vishwanathan. Hash Kernels.
- [Shi2] Qinfeng Shi; James Petterson; John Langford; Alex Smola; Alex Strehl; Gideon Dror; Vishy Vishwanathan. Hash Kernels for Structured Data.
- [Singh-Memoli-Carlson] Gruheet Singh; Facundo Memoli; Gunnar Carlson. Topological Methods for the Analysis of High Dimensional Data Sets and 3D Object Recognition.
- [Swamidass-Baldi] S Joshua Swamidass; Pierre Baldi. Bounds and algorithms for fast exact searches of chemical fingerprints in linear and sublinear time.
- [Ugander] Johan Ugander; Brian Karrer; Lars Backstrom; Cameron Marlow. The Anatomy of the Facebook Social Graph.
- [Valiant] Leslie Valiant. A Theory of the Learnable.
- [van der Maaten-Hinton] Laurens van der Maaten; Geoffrey Hinton. Visualizing Data using t-SNE.
- [van de Maaten] Laurens van der Maaten. Learning a Parametric Embedding by Preserving Local Structure.
- [Vitanyi] Rudi Cilibrasi and Paul M.B. Vitanyi. Clustering by Compression
- [Ward] J.H Ward Jr. Hierarchical Grouping to Optimize an Objective Function.
- [Weinberger] Killian Weinberger; Anirban Dasgupta; John Langford; Alex Smola; Josh Attenberg. Feature Hashing for Large Scale Multitask Learning.
- [Weiss-Ng-Jordan] Yair Weiss; Andrew Ng; Michael Jordan. On Spectral Clustering Analysis and Algorithm.
- [Wickham] Hadley Wickham. (2004). Tidy Data.
- [Witten-Bell] I.H. Witten and T.C. Bell. The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression.
- [Wolpert] D.H. Wolpert; W.G. Macready. No Free Lunch Theorems for Optimization.
- [Yan-Huang-Jordan] Donghui Yan; Lin Huang; Michael Jordan. Fast Approximate Spectral Clustering.
- [Panther] Jing Zhang; Jie Tang; Cong Ma; Hanghang Tong; Yu Jing; Juanzi Li. Panther Fast Top K Similarity Search in Large Networks
- [Zhou-Li-Andersen-Smola] Li Zhou; Mu Li; David Andersen; Alexander Smola. Cuckoo Linear Algebra.