An

Introduction

to

Information

Retrieval

Draft of April 1, 2009

Online edition (c)2009 Cambridge UP

Online edition (c)2009 Cambridge UP

An

Introduction to

Information Retrieval

Christopher D. Manning Prabhakar Raghavan

Hinrich Schütze

Cambridge University Press Cambridge, England

Online edition (c)2009 Cambridge UP

DRAFT!

DO NOT DISTRIBUTE WITHOUT PRIOR PERMISSION © 2009 Cambridge University Press

By Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze Printed on April 1, 2009

Website: http://www.informationretrieval.org/

Comments, corrections, and other feedback most welcome at: informationretrieval@yahoogroups.com

Online edition (c)2009 Cambridge UP

DRAFT! © April 1, 2009 Cambridge University Press. Feedback welcome. *v* ***Brief Contents***

*1 Boolean retrieval* 1

*2 The term vocabulary and postings lists* 19

*3 Dictionaries and tolerant retrieval* 49

*4 Index construction* 67

*5 Index compression* 85

*6 Scoring, term weighting and the vector space model* 109 *7 Computing scores in a complete search system* 135

*8 Evaluation in information retrieval* 151

*9 Relevance feedback and query expansion* 177

*10 XML retrieval* 195

*11 Probabilistic information retrieval* 219

*12 Language models for information retrieval* 237

*13 Text classification and Naive Bayes* 253

*14 Vector space classification* 289

*15 Support vector machines and machine learning on documents* 319 *16 Flat clustering* 349

*17 Hierarchical clustering* 377

*18 Matrix decompositions and latent semantic indexing* 403 *19 Web search basics* 421

*20 Web crawling and indexes* 443

*21 Link analysis* 461

Online edition (c)2009 Cambridge UP

Online edition (c)2009 Cambridge UP

DRAFT! © April 1, 2009 Cambridge University Press. Feedback welcome. *vii* ***Contents***

***List of Tables* xv**

***List of Figures* xix**

***Table of Notation* xxvii**

***Preface* xxxi**

**1 *Boolean retrieval* 1**

1.1 An example information retrieval problem 3

1.2 A first take at building an inverted index 6

1.3 Processing Boolean queries 10

1.4 The extended Boolean model versus ranked retrieval 14 1.5 References and further reading 17

**2 *The term vocabulary and postings lists* 19**

2.1 Document delineation and character sequence decoding 19 2.1.1 Obtaining the character sequence in a document 19 2.1.2 Choosing a document unit 20

2.2 Determining the vocabulary of terms 22

2.2.1 Tokenization 22

2.2.2 Dropping common terms: stop words 27

2.2.3 Normalization (equivalence classing of terms) 28 2.2.4 Stemming and lemmatization 32

2.3 Faster postings list intersection via skip pointers 36 2.4 Positional postings and phrase queries 39

2.4.1 Biword indexes 39

2.4.2 Positional indexes 41

2.4.3 Combination schemes 43

2.5 References and further reading 45

Online edition (c)2009 Cambridge UP

*viii Contents*

**3 *Dictionaries and tolerant retrieval* 49**

3.1 Search structures for dictionaries 49

3.2 Wildcard queries 51

3.2.1 General wildcard queries 53

3.2.2 *k*-gram indexes for wildcard queries 54

3.3 Spelling correction 56

3.3.1 Implementing spelling correction 57

3.3.2 Forms of spelling correction 57

3.3.3 Edit distance 58

3.3.4 *k*-gram indexes for spelling correction 60

3.3.5 Context sensitive spelling correction 62

3.4 Phonetic correction 63

3.5 References and further reading 65

**4 *Index construction* 67**

4.1 Hardware basics 68

4.2 Blocked sort-based indexing 69

4.3 Single-pass in-memory indexing 73

4.4 Distributed indexing 74

4.5 Dynamic indexing 78

4.6 Other types of indexes 80

4.7 References and further reading 83

**5 *Index compression* 85**

5.1 Statistical properties of terms in information retrieval 86

5.1.1 Heaps’ law: Estimating the number of terms 88

5.1.2 Zipf’s law: Modeling the distribution of terms 89

5.2 Dictionary compression 90

5.2.1 Dictionary as a string 91

5.2.2 Blocked storage 92

5.3 Postings file compression 95

5.3.1 Variable byte codes 96

5.3.2 *γ* codes 98

5.4 References and further reading 105

**6 *Scoring, term weighting and the vector space model* 109**

6.1 Parametric and zone indexes 110

6.1.1 Weighted zone scoring 112

6.1.2 Learning weights 113

6.1.3 The optimal weight *g* 115

6.2 Term frequency and weighting 117

6.2.1 Inverse document frequency 117

6.2.2 Tf-idf weighting 118

Online edition (c)2009 Cambridge UP

*Contents ix*

6.3 The vector space model for scoring 120

6.3.1 Dot products 120

6.3.2 Queries as vectors 123

6.3.3 Computing vector scores 124

6.4 Variant tf-idf functions 126

6.4.1 Sublinear tf scaling 126

6.4.2 Maximum tf normalization 127

6.4.3 Document and query weighting schemes 128 6.4.4 Pivoted normalized document length 129

6.5 References and further reading 133

**7 *Computing scores in a complete search system* 135** 7.1 Efficient scoring and ranking 135

7.1.1 Inexact top *K* document retrieval 137

7.1.2 Index elimination 137

7.1.3 Champion lists 138

7.1.4 Static quality scores and ordering 138

7.1.5 Impact ordering 140

7.1.6 Cluster pruning 141

7.2 Components of an information retrieval system 143 7.2.1 Tiered indexes 143

7.2.2 Query-term proximity 144

7.2.3 Designing parsing and scoring functions 145 7.2.4 Putting it all together 146

7.3 Vector space scoring and query operator interaction 147 7.4 References and further reading 149

**8 *Evaluation in information retrieval* 151**

8.1 Information retrieval system evaluation 152

8.2 Standard test collections 153

8.3 Evaluation of unranked retrieval sets 154

8.4 Evaluation of ranked retrieval results 158

8.5 Assessing relevance 164

8.5.1 Critiques and justifications of the concept of

relevance 166

8.6 A broader perspective: System quality and user utility 168 8.6.1 System issues 168

8.6.2 User utility 169

8.6.3 Refining a deployed system 170

8.7 Results snippets 170

8.8 References and further reading 173

**9 *Relevance feedback and query expansion* 177**

Online edition (c)2009 Cambridge UP

*x Contents*

9.1 Relevance feedback and pseudo relevance feedback 178

9.1.1 The Rocchio algorithm for relevance feedback 178

9.1.2 Probabilistic relevance feedback 183

9.1.3 When does relevance feedback work? 183

9.1.4 Relevance feedback on the web 185

9.1.5 Evaluation of relevance feedback strategies 186

9.1.6 Pseudo relevance feedback 187

9.1.7 Indirect relevance feedback 187

9.1.8 Summary 188

9.2 Global methods for query reformulation 189

9.2.1 Vocabulary tools for query reformulation 189

9.2.2 Query expansion 189

9.2.3 Automatic thesaurus generation 192

9.3 References and further reading 193

**10 *XML retrieval* 195**

10.1 Basic XML concepts 197

10.2 Challenges in XML retrieval 201

10.3 A vector space model for XML retrieval 206

10.4 Evaluation of XML retrieval 210

10.5 Text-centric vs. data-centric XML retrieval 214

10.6 References and further reading 216

10.7 Exercises 217

**11 *Probabilistic information retrieval* 219**

11.1 Review of basic probability theory 220

11.2 The Probability Ranking Principle 221

11.2.1 The 1/0 loss case 221

11.2.2 The PRP with retrieval costs 222

11.3 The Binary Independence Model 222

11.3.1 Deriving a ranking function for query terms 224

11.3.2 Probability estimates in theory 226

11.3.3 Probability estimates in practice 227

11.3.4 Probabilistic approaches to relevance feedback 228

11.4 An appraisal and some extensions 230

11.4.1 An appraisal of probabilistic models 230

11.4.2 Tree-structured dependencies between terms 231

11.4.3 Okapi BM25: a non-binary model 232

11.4.4 Bayesian network approaches to IR 234

11.5 References and further reading 235

**12 *Language models for information retrieval* 237**

12.1 Language models 237

Online edition (c)2009 Cambridge UP

*Contents xi*

12.1.1 Finite automata and language models 237

12.1.2 Types of language models 240

12.1.3 Multinomial distributions over words 241

12.2 The query likelihood model 242

12.2.1 Using query likelihood language models in IR 242 12.2.2 Estimating the query generation probability 243 12.2.3 Ponte and Croft’s Experiments 246

12.3 Language modeling versus other approaches in IR 248 12.4 Extended language modeling approaches 250

12.5 References and further reading 252

**13 *Text classification and Naive Bayes* 253**

13.1 The text classification problem 256

13.2 Naive Bayes text classification 258

13.2.1 Relation to multinomial unigram language model 262 13.3 The Bernoulli model 263

13.4 Properties of Naive Bayes 265

13.4.1 A variant of the multinomial model 270

13.5 Feature selection 271

13.5.1 Mutual information 272

13.5.2 *χ*2 Feature selection 275

13.5.3 Frequency-based feature selection 277

13.5.4 Feature selection for multiple classifiers 278

13.5.5 Comparison of feature selection methods 278 13.6 Evaluation of text classification 279

13.7 References and further reading 286

**14 *Vector space classification* 289**

14.1 Document representations and measures of relatedness in vector spaces 291

14.2 Rocchio classification 292

14.3 *k* nearest neighbor 297

14.3.1 Time complexity and optimality of kNN 299

14.4 Linear versus nonlinear classifiers 301

14.5 Classification with more than two classes 306

14.6 The bias-variance tradeoff 308

14.7 References and further reading 314

14.8 Exercises 315

**15 *Support vector machines and machine learning on documents* 319** 15.1 Support vector machines: The linearly separable case 320 15.2 Extensions to the SVM model 327

15.2.1 Soft margin classification 327

Online edition (c)2009 Cambridge UP

*xii Contents*

15.2.2 Multiclass SVMs 330

15.2.3 Nonlinear SVMs 330

15.2.4 Experimental results 333

15.3 Issues in the classification of text documents 334

15.3.1 Choosing what kind of classifier to use 335

15.3.2 Improving classifier performance 337

15.4 Machine learning methods in ad hoc information retrieval 341

15.4.1 A simple example of machine-learned scoring 341

15.4.2 Result ranking by machine learning 344

15.5 References and further reading 346

**16 *Flat clustering* 349**

16.1 Clustering in information retrieval 350

16.2 Problem statement 354

16.2.1 Cardinality – the number of clusters 355

16.3 Evaluation of clustering 356

16.4 *K*-means 360

16.4.1 Cluster cardinality in *K*-means 365

16.5 Model-based clustering 368

16.6 References and further reading 372

16.7 Exercises 374

**17 *Hierarchical clustering* 377**

17.1 Hierarchical agglomerative clustering 378

17.2 Single-link and complete-link clustering 382

17.2.1 Time complexity of HAC 385

17.3 Group-average agglomerative clustering 388

17.4 Centroid clustering 391

17.5 Optimality of HAC 393

17.6 Divisive clustering 395

17.7 Cluster labeling 396

17.8 Implementation notes 398

17.9 References and further reading 399

17.10 Exercises 401

**18 *Matrix decompositions and latent semantic indexing* 403**

18.1 Linear algebra review 403

18.1.1 Matrix decompositions 406

18.2 Term-document matrices and singular value

decompositions 407

18.3 Low-rank approximations 410

18.4 Latent semantic indexing 412

18.5 References and further reading 417

Online edition (c)2009 Cambridge UP

*Contents xiii*

**19 *Web search basics* 421**

19.1 Background and history 421

19.2 Web characteristics 423

19.2.1 The web graph 425

19.2.2 Spam 427

19.3 Advertising as the economic model 429

19.4 The search user experience 432

19.4.1 User query needs 432

19.5 Index size and estimation 433

19.6 Near-duplicates and shingling 437

19.7 References and further reading 441

**20 *Web crawling and indexes* 443**

20.1 Overview 443

20.1.1 Features a crawler *must* provide 443

20.1.2 Features a crawler *should* provide 444

20.2 Crawling 444

20.2.1 Crawler architecture 445

20.2.2 DNS resolution 449

20.2.3 The URL frontier 451

20.3 Distributing indexes 454

20.4 Connectivity servers 455

20.5 References and further reading 458

**21 *Link analysis* 461**

21.1 The Web as a graph 462

21.1.1 Anchor text and the web graph 462

21.2 PageRank 464

21.2.1 Markov chains 465

21.2.2 The PageRank computation 468

21.2.3 Topic-specific PageRank 471

21.3 Hubs and Authorities 474

21.3.1 Choosing the subset of the Web 477

21.4 References and further reading 480

***Bibliography* 483**

***Author Index* 519**

Online edition (c)2009 Cambridge UP

Online edition (c)2009 Cambridge UP

DRAFT! © April 1, 2009 Cambridge University Press. Feedback welcome. *xv* ***List of Tables***

4.1 Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The

transfer time per byte is the rate of transfer from disk to

memory when the head is in the right position. 68 4.2 Collection statistics for Reuters-RCV1. Values are rounded for the computations in this book. The unrounded values are: 806,791 documents, 222 tokens per document, 391,523

(distinct) terms, 6.04 bytes per token with spaces and

punctuation, 4.5 bytes per token without spaces and

punctuation, 7.5 bytes per term, and 96,969,056 tokens. The numbers in this table correspond to the third line (“case

folding”) in Table 5.1 (page 87). 70 4.3 The five steps in constructing an index for Reuters-RCV1 in blocked sort-based indexing. Line numbers refer to Figure 4.2. 82 4.4 Collection statistics for a large collection. 82

5.1 The effect of preprocessing on the number of terms, nonpositional postings, and tokens for Reuters-RCV1. “∆%” indicates the reduction in size from the previous line, except that “30 stop words” and “150 stop words” both use “case folding” as their reference line. “T%” is the cumulative

(“total”) reduction from unfiltered. We performed stemming with the Porter stemmer (Chapter 2, page 33). 87 5.2 Dictionary compression for Reuters-RCV1. 95 5.3 Encoding gaps instead of document IDs. For example, we store gaps 107, 5, 43, . . . , instead of docIDs 283154, 283159, 283202, . . . for computer. The first docID is left unchanged (only shown for arachnocentric). 96 5.4 VB encoding. 97

Online edition (c)2009 Cambridge UP

*xvi List of Tables*

5.5 Some examples of unary and *γ* codes. Unary codes are only

shown for the smaller numbers. Commas in *γ* codes are for

readability only and are not part of the actual codes. 98

5.6 Index and dictionary compression for Reuters-RCV1. The

compression ratio depends on the proportion of actual text in

the collection. Reuters-RCV1 contains a large amount of XML

markup. Using the two best compression schemes, *γ*

encoding and blocking with front coding, the ratio

compressed index to collection size is therefore especially

small for Reuters-RCV1: (101 + 5.9)/3600 ≈ 0.03. 103

5.7 Two gap sequences to be merged in blocked sort-based

indexing 105

6.1 Cosine computation for Exercise 6.19. 132

8.1 Calculation of 11-point Interpolated Average Precision. 159 8.2 Calculating the kappa statistic. 165

10.1 RDB (relational database) search, unstructured information

retrieval and structured information retrieval. 196

10.2 INEX 2002 collection statistics. 211 10.3 INEX 2002 results of the vector space model in Section 10.3 for

content-and-structure (CAS) queries and the quantization

function **Q**. 213

10.4 A comparison of content-only and full-structure search in

INEX 2003/2004. 214

13.1 Data for parameter estimation examples. 261 13.2 Training and test times for NB. 261 13.3 Multinomial versus Bernoulli model. 268 13.4 Correct estimation implies accurate prediction, but accurate

prediction does not imply correct estimation. 269

13.5 A set of documents for which the NB independence

assumptions are problematic. 270

13.6 Critical values of the *χ*2 distribution with one degree of

freedom. For example, if the two events are independent,

then *P*(*X*2 > 6.63) < 0.01. So for *X*2 > 6.63 the assumption of

independence can be rejected with 99% confidence. 277

13.7 The ten largest classes in the Reuters-21578 collection with

number of documents in training and test sets. 280

Online edition (c)2009 Cambridge UP

*List of Tables xvii*

13.8 Macro- and microaveraging. “Truth” is the true class and “call” the decision of the classifier. In this example,

macroaveraged precision is

[10/(10 + 10) + 90/(10 + 90)]/2 = (0.5 + 0.9)/2 = 0.7.

Microaveraged precision is 100/(100 + 20) ≈ 0.83. 282 13.9 Text classification effectiveness numbers on Reuters-21578 for F1(in percent). Results from Li and Yang (2003) (a), Joachims (1998) (b: kNN) and Dumais et al. (1998) (b: NB, Rocchio,

trees, SVM). 282 13.10 Data for parameter estimation exercise. 284

14.1 Vectors and class centroids for the data in Table 13.1. 294 14.2 Training and test times for Rocchio classification. 296 14.3 Training and test times for kNN classification. 299 14.4 A linear classifier. 303 14.5 A confusion matrix for Reuters-21578. 308

15.1 Training and testing complexity of various classifiers including SVMs. 329 15.2 SVM classifier break-even F1from (Joachims 2002a, p. 114). 334 15.3 Training examples for machine-learned scoring. 342

16.1 Some applications of clustering in information retrieval. 351 16.2 The four external evaluation measures applied to the clustering in Figure 16.4. 357 16.3 The EM clustering algorithm. 371

17.1 Comparison of HAC algorithms. 395 17.2 Automatically computed cluster labels. 397

Online edition (c)2009 Cambridge UP

Online edition (c)2009 Cambridge UP

DRAFT! © April 1, 2009 Cambridge University Press. Feedback welcome. *xix* ***List of Figures***

1.1 A term-document incidence matrix. 4 1.2 Results from Shakespeare for the query Brutus AND Caesar AND NOT Calpurnia. 5 1.3 The two parts of an inverted index. 7 1.4 Building an index by sorting and grouping. 8 1.5 Intersecting the postings lists for Brutus and Calpurnia from Figure 1.3. 10 1.6 Algorithm for the intersection of two postings lists *p*1 and *p*2. 11 1.7 Algorithm for conjunctive queries that returns the set of documents containing each term in the input list of terms. 12

2.1 An example of a vocalized Modern Standard Arabic word. 21 2.2 The conceptual linear order of characters is not necessarily the order that you see on the page. 21 2.3 The standard unsegmented form of Chinese text using the simplified characters of mainland China. 26 2.4 Ambiguities in Chinese word segmentation. 26 2.5 A stop list of 25 semantically non-selective words which are common in Reuters-RCV1. 26 2.6 An example of how asymmetric expansion of query terms can usefully model users’ expectations. 28 2.7 Japanese makes use of multiple intermingled writing systems and, like Chinese, does not segment words. 31 2.8 A comparison of three stemming algorithms on a sample text. 34 2.9 Postings lists with skip pointers. 36 2.10 Postings lists intersection with skip pointers. 37 2.11 Positional index example. 41 2.12 An algorithm for proximity intersection of postings lists *p*1 and *p*2. 42

Online edition (c)2009 Cambridge UP

*xx List of Figures*

3.1 A binary search tree. 51 3.2 A B-tree. 52 3.3 A portion of a permuterm index. 54 3.4 Example of a postings list in a 3-gram index. 55 3.5 Dynamic programming algorithm for computing the edit

distance between strings *s*1 and *s*2. 59

3.6 Example Levenshtein distance computation. 59 3.7 Matching at least two of the three 2-grams in the query bord. 61

4.1 Document from the Reuters newswire. 70 4.2 Blocked sort-based indexing. 71 4.3 Merging in blocked sort-based indexing. 72 4.4 Inversion of a block in single-pass in-memory indexing 73 4.5 An example of distributed indexing with MapReduce.

Adapted from Dean and Ghemawat (2004). 76

4.6 Map and reduce functions in MapReduce. 77 4.7 Logarithmic merging. Each token (termID,docID) is initially

added to in-memory index *Z*0 by LMERGEADDTOKEN.

LOGARITHMICMERGE initializes *Z*0 and *indexes*. 79

4.8 A user-document matrix for access control lists. Element (*i*, *j*)

is 1 if user *i* has access to document *j* and 0 otherwise. During

query processing, a user’s access postings list is intersected

with the results list returned by the text part of the index. 81

5.1 Heaps’ law. 88 5.2 Zipf’s law for Reuters-RCV1. 90 5.3 Storing the dictionary as an array of fixed-width entries. 91 5.4 Dictionary-as-a-string storage. 92 5.5 Blocked storage with four terms per block. 93 5.6 Search of the uncompressed dictionary (a) and a dictionary

compressed by blocking with *k* = 4 (b). 94

5.7 Front coding. 94 5.8 VB encoding and decoding. 97 5.9 Entropy *H*(*P*) as a function of *P*(*x*1) for a sample space with

two outcomes *x*1 and *x*2. 100

5.10 Stratification of terms for estimating the size of a *γ* encoded

inverted index. 102

6.1 Parametric search. 111 6.2 Basic zone index 111 6.3 Zone index in which the zone is encoded in the postings

rather than the dictionary. 111

Online edition (c)2009 Cambridge UP

*List of Figures xxi*

6.4 Algorithm for computing the weighted zone score from two postings lists. 113 6.5 An illustration of training examples. 115 6.6 The four possible combinations of *sT* and *sB*. 115 6.7 Collection frequency (cf) and document frequency (df) behave differently, as in this example from the Reuters collection. 118 6.8 Example of idf values. 119 6.9 Table of tf values for Exercise 6.10. 120 6.10 Cosine similarity illustrated. 121 6.11 Euclidean normalized tf values for documents in Figure 6.9. 122 6.12 Term frequencies in three novels. 122 6.13 Term vectors for the three novels of Figure 6.12. 123 6.14 The basic algorithm for computing vector space scores. 125 6.15 SMART notation for tf-idf variants. 128 6.16 Pivoted document length normalization. 130 6.17 Implementing pivoted document length normalization by linear scaling. 131

7.1 A faster algorithm for vector space scores. 136 7.2 A static quality-ordered index. 139 7.3 Cluster pruning. 142 7.4 Tiered indexes. 144 7.5 A complete search system. 147

8.1 Graph comparing the harmonic mean to other means. 157 8.2 Precision/recall graph. 158 8.3 Averaged 11-point precision/recall graph across 50 queries

for a representative TREC system. 160 8.4 The ROC curve corresponding to the precision-recall curve in Figure 8.2. 162 8.5 An example of selecting text for a dynamic snippet. 172

9.1 Relevance feedback searching over images. 179 9.2 Example of relevance feedback on a text collection. 180 9.3 The Rocchio optimal query for separating relevant and

nonrelevant documents. 181 9.4 An application of Rocchio’s algorithm. 182 9.5 Results showing pseudo relevance feedback greatly

improving performance. 187 9.6 An example of query expansion in the interface of the Yahoo! web search engine in 2006. 190 9.7 Examples of query expansion via the PubMed thesaurus. 191 9.8 An example of an automatically generated thesaurus. 192

Online edition (c)2009 Cambridge UP

*xxii List of Figures*

10.1 An XML document. 198 10.2 The XML document in Figure 10.1 as a simplified DOM object. 198 10.3 An XML query in NEXI format and its partial representation

as a tree. 199

10.4 Tree representation of XML documents and queries. 200 10.5 Partitioning an XML document into non-overlapping

indexing units. 202

10.6 Schema heterogeneity: intervening nodes and mismatched

names. 204

10.7 A structural mismatch between two queries and a document. 206 10.8 A mapping of an XML document (left) to a set of lexicalized

subtrees (right). 207

10.9 The algorithm for scoring documents with SIMNOMERGE. 209 10.10 Scoring of a query with one structural term in SIMNOMERGE. 209 10.11 Simplified schema of the documents in the INEX collection. 211

11.1 A tree of dependencies between terms. 232

12.1 A simple finite automaton and some of the strings in the

language it generates. 238

12.2 A one-state finite automaton that acts as a unigram language

model. 238

12.3 Partial specification of two unigram language models. 239 12.4 Results of a comparison of tf-idf with language modeling

(LM) term weighting by Ponte and Croft (1998). 247

12.5 Three ways of developing the language modeling approach:

(a) query likelihood, (b) document likelihood, and (c) model

comparison. 250

13.1 Classes, training set, and test set in text classification . 257 13.2 Naive Bayes algorithm (multinomial model): Training and

testing. 260

13.3 NB algorithm (Bernoulli model): Training and testing. 263 13.4 The multinomial NB model. 266 13.5 The Bernoulli NB model. 267 13.6 Basic feature selection algorithm for selecting the *k* best features. 271 13.7 Features with high mutual information scores for six

Reuters-RCV1 classes. 274

13.8 Effect of feature set size on accuracy for multinomial and

Bernoulli models. 275

13.9 A sample document from the Reuters-21578 collection. 281 14.1 Vector space classification into three classes. 290

Online edition (c)2009 Cambridge UP

*List of Figures xxiii*

14.2 Projections of small areas of the unit sphere preserve distances. 291 14.3 Rocchio classification. 293 14.4 Rocchio classification: Training and testing. 295 14.5 The multimodal class “a” consists of two different clusters

(small upper circles centered on X’s). 295 14.6 Voronoi tessellation and decision boundaries (double lines) in 1NN classification. 297 14.7 kNN training (with preprocessing) and testing. 298 14.8 There are an infinite number of hyperplanes that separate two linearly separable classes. 301 14.9 Linear classification algorithm. 302 14.10 A linear problem with noise. 304 14.11 A nonlinear problem. 305 14.12 *J* hyperplanes do not divide space into *J* disjoint regions. 307 14.13 Arithmetic transformations for the bias-variance decomposition. 310 14.14 Example for differences between Euclidean distance, dot product similarity and cosine similarity. 316 14.15 A simple non-separable set of points. 317

15.1 The support vectors are the 5 points right up against the margin of the classifier. 320 15.2 An intuition for large-margin classification. 321 15.3 The geometric margin of a point (*r*) and a decision boundary (*ρ*). 323 15.4 A tiny 3 data point training set for an SVM. 325 15.5 Large margin classification with slack variables. 327 15.6 Projecting data that is not linearly separable into a higher dimensional space can make it linearly separable. 331 15.7 A collection of training examples. 343

16.1 An example of a data set with a clear cluster structure. 349 16.2 Clustering of search results to improve recall. 352 16.3 An example of a user session in Scatter-Gather. 353 16.4 Purity as an external evaluation criterion for cluster quality. 357 16.5 The *K*-means algorithm. 361 16.6 A *K*-means example for *K* = 2 in **R**2. 362 16.7 The outcome of clustering in *K*-means depends on the initial

seeds. 364 16.8 Estimated minimal residual sum of squares as a function of the number of clusters in *K*-means. 366

17.1 A dendrogram of a single-link clustering of 30 documents from Reuters-RCV1. 379 17.2 A simple, but inefficient HAC algorithm. 381

Online edition (c)2009 Cambridge UP

*xxiv List of Figures*

17.3 The different notions of cluster similarity used by the four

HAC algorithms. 381

17.4 A single-link (left) and complete-link (right) clustering of

eight documents. 382

17.5 A dendrogram of a complete-link clustering. 383 17.6 Chaining in single-link clustering. 384 17.7 Outliers in complete-link clustering. 385 17.8 The priority-queue algorithm for HAC. 386 17.9 Single-link clustering algorithm using an NBM array. 387 17.10 Complete-link clustering is not best-merge persistent. 388 17.11 Three iterations of centroid clustering. 391 17.12 Centroid clustering is not monotonic. 392

18.1 Illustration of the singular-value decomposition. 409 18.2 Illustration of low rank approximation using the

singular-value decomposition. 411

18.3 The documents of Example 18.4 reduced to two dimensions

in (*V*′)*T*. 416

18.4 Documents for Exercise 18.11. 418 18.5 Glossary for Exercise 18.11. 418

19.1 A dynamically generated web page. 425 19.2 Two nodes of the web graph joined by a link. 425 19.3 A sample small web graph. 426 19.4 The bowtie structure of the Web. 427 19.5 Cloaking as used by spammers. 428 19.6 Search advertising triggered by query keywords. 431 19.7 The various components of a web search engine. 434 19.8 Illustration of shingle sketches. 439 19.9 Two sets *Sj*1and *Sj*2; their Jaccard coefficient is 2/5. 440

20.1 The basic crawler architecture. 446 20.2 Distributing the basic crawl architecture. 449 20.3 The URL frontier. 452 20.4 Example of an auxiliary hosts-to-back queues table. 453 20.5 A lexicographically ordered set of URLs. 456 20.6 A four-row segment of the table of links. 457

21.1 The random surfer at node A proceeds with probability 1/3 to

each of B, C and D. 464

21.2 A simple Markov chain with three states; the numbers on the

links indicate the transition probabilities. 466

21.3 The sequence of probability vectors. 469

Online edition (c)2009 Cambridge UP

*List of Figures xxv*

21.4 A small web graph. 470 21.5 Topic-specific PageRank. 472 21.6 A sample run of HITS on the query japan elementary schools. 479 21.7 Web graph for Exercise 21.22. 480

Online edition (c)2009 Cambridge UP

Online edition (c)2009 Cambridge UP

DRAFT! © April 1, 2009 Cambridge University Press. Feedback welcome. *xxvii* ***Table of Notation***

**Symbol Page Meaning**

*γ* p. 98 *γ* code

*γ* p. 256 Classification or clustering function: *γ*(*d*) is *d*’s class or cluster

Γ p. 256 Supervised learning method in Chapters 13 and 14: Γ(**D**) is the classification function *γ* learned from

training set **D**

*λ* p. 404 Eigenvalue

~*µ*(.) p. 292 Centroid of a class (in Rocchio classification) or a cluster (in *K*-means and centroid clustering)

Φ p. 114 Training example

*σ* p. 408 Singular value

Θ(·) p. 11 A tight bound on the complexity of an algorithm *ω*, *ωk* p. 357 Cluster in clustering

Ω p. 357 Clustering or set of clusters {*ω*1, . . . , *ωK*} arg max*xf*(*x*) p. 181 The value of *x* for which *f* reaches its maximum arg min*xf*(*x*) p. 181 The value of *x* for which *f* reaches its minimum *c*, *cj* p. 256 Class or category in classification

cf*t* p. 89 The collection frequency of term *t* (the total number of times the term appears in the document collec

tion)

**C** p. 256 Set {*c*1, . . . , *cJ*} of all classes

*C* p. 268 A random variable that takes as values members of **C**

Online edition (c)2009 Cambridge UP

*xxviii Table of Notation*

*C* p. 403 Term-document matrix

*d* p. 4 Index of the *d*th document in the collection *D*

*d* p. 71 A document

*d*~,~*q* p. 181 Document vector, query vector

*D* p. 354 Set {*d*1, . . . , *dN*} of all documents

*Dc* p. 292 Set of documents that is in class *c*

**D** p. 256 Set {h*d*1, *c*1i, . . . ,h*dN*, *cN*i} of all labeled documents in Chapters 13–15

df*t* p. 118 The document frequency of term *t* (the total number of documents in the collection the term appears in)

*H* p. 99 Entropy

*HM* p. 101 *M*th harmonic number

*I*(*X*;*Y*) p. 272 Mutual information of random variables *X* and *Y*

idf*t* p. 118 Inverse document frequency of term *t*

*J* p. 256 Number of classes

*k* p. 290 Top *k* items from a set, e.g., *k* nearest neighbors in kNN, top *k* retrieved documents, top *k* selected fea

tures from the vocabulary *V*

*k* p. 54 Sequence of *k* characters

*K* p. 354 Number of clusters

*Ld* p. 233 Length of document *d* (in tokens)

*L*a p. 262 Length of the test document (or application docu ment) in tokens

*L*ave p. 70 Average length of a document (in tokens)

*M* p. 5 Size of the vocabulary (|*V*|)

*M*a p. 262 Size of the vocabulary of the test document (or ap plication document)

*M*ave p. 78 Average size of the vocabulary in a document in the collection

*Md* p. 237 Language model for document *d*

*N* p. 4 Number of documents in the retrieval or training collection

*Nc* p. 259 Number of documents in class *c*

*N*(*ω*) p. 298 Number of times the event *ω* occurred

Online edition (c)2009 Cambridge UP

*Table of Notation xxix*

*O*(·) p. 11 A bound on the complexity of an algorithm *O*(·) p. 221 The odds of an event

*P* p. 155 Precision

*P*(·) p. 220 Probability

*P* p. 465 Transition probability matrix

*q* p. 59 A query

*R* p. 155 Recall

*si* p. 58 A string

*si* p. 112 Boolean values for zone scoring

sim(*d*1, *d*2) p. 121 Similarity score for documents *d*1, *d*2 *T* p. 43 Total number of tokens in the document collection

*Tct* p. 259 Number of occurrences of word *t* in documents of class *c*

*t* p. 4 Index of the *t*th term in the vocabulary *V t* p. 61 A term in the vocabulary

tf*t*,*d* p. 117 The term frequency of term *t* in document *d* (the to tal number of occurrences of *t* in *d*)

*Ut* p. 266 Random variable taking values 0 (term *t* is present) and 1 (*t* is not present)

*V* p. 208 Vocabulary of terms {*t*1, . . . , *tM*} in a collection (a.k.a. the lexicon)

~*v*(*d*) p. 122 Length-normalized document vector *V*~ (*d*) p. 120 Vector of document *d*, not length-normalized wf*t*,*d* p. 125 Weight of term *t* in document *d*

*w* p. 112 A weight, for example for zones or terms

*w*~T~*x* = *b* p. 293 Hyperplane; *w*~ is the normal vector of the hyper plane and *wi* component *i* of *w*~

~*x* p. 222 Term incidence vector ~*x* = (*x*1, . . . , *xM*); more gen erally: document feature representation

*X* p. 266 Random variable taking values in *V*, the vocabulary (e.g., at a given position *k* in a document)

**X** p. 256 Document space in text classification |*A*| p. 61 Set cardinality: the number of members of set *A* |*S*| p. 404 Determinant of the square matrix *S*

Online edition (c)2009 Cambridge UP

*xxx Table of Notation*

|*si*| p. 58 Length in characters of string *si*

|~*x*| p. 139 Length of vector ~*x*

|~*x* −~*y*| p. 131 Euclidean distance of ~*x* and ~*y* (which is the length of (~*x* −~*y*))

Online edition (c)2009 Cambridge UP

DRAFT! © April 1, 2009 Cambridge University Press. Feedback welcome. *xxxi* ***Preface***

As recently as the 1990s, studies showed that most people preferred getting information from other people rather than from information retrieval sys tems. Of course, in that time period, most people also used human travel agents to book their travel. However, during the last decade, relentless opti mization of information retrieval effectiveness has driven web search engines to new quality levels where most people are satisfied most of the time, and web search has become a standard and often preferred source of information finding. For example, the 2004 Pew Internet Survey (Fallows 2004) found that “92% of Internet users say the Internet is a good place to go for getting everyday information.” To the surprise of many, the field of information re trieval has moved from being a primarily academic discipline to being the basis underlying most people’s preferred means of information access. This book presents the scientific underpinnings of this field, at a level accessible to graduate students as well as advanced undergraduates.

Information retrieval did not begin with the Web. In response to various challenges of providing information access, the field of information retrieval evolved to give principled approaches to searching various forms of con tent. The field began with scientific publications and library records, but soon spread to other forms of content, particularly those of information pro fessionals, such as journalists, lawyers, and doctors. Much of the scientific research on information retrieval has occurred in these contexts, and much of the continued practice of information retrieval deals with providing access to unstructured information in various corporate and governmental domains, and this work forms much of the foundation of our book.

Nevertheless, in recent years, a principal driver of innovation has been the World Wide Web, unleashing publication at the scale of tens of millions of content creators. This explosion of published information would be moot if the information could not be found, annotated and analyzed so that each user can quickly find information that is both relevant and comprehensive for their needs. By the late 1990s, many people felt that continuing to index

Online edition (c)2009 Cambridge UP

*xxxii Preface*

the whole Web would rapidly become impossible, due to the Web’s expo nential growth in size. But major scientific innovations, superb engineering, the rapidly declining price of computer hardware, and the rise of a commer cial underpinning for web search have all conspired to power today’s major search engines, which are able to provide high-quality results within subsec ond response times for hundreds of millions of searches a day over billions of web pages.

**Book organization and course development**

This book is the result of a series of courses we have taught at Stanford Uni versity and at the University of Stuttgart, in a range of durations including a single quarter, one semester and two quarters. These courses were aimed at early-stage graduate students in computer science, but we have also had enrollment from upper-class computer science undergraduates, as well as students from law, medical informatics, statistics, linguistics and various en gineering disciplines. The key design principle for this book, therefore, was to cover what we believe to be important in a one-term graduate course on information retrieval. An additional principle is to build each chapter around material that we believe can be covered in a single lecture of 75 to 90 minutes. The first eight chapters of the book are devoted to the basics of informa

tion retrieval, and in particular the heart of search engines; we consider this material to be core to any course on information retrieval. Chapter 1 in troduces inverted indexes, and shows how simple Boolean queries can be processed using such indexes. Chapter 2 builds on this introduction by de tailing the manner in which documents are preprocessed before indexing and by discussing how inverted indexes are augmented in various ways for functionality and speed. Chapter 3 discusses search structures for dictionar ies and how to process queries that have spelling errors and other imprecise matches to the vocabulary in the document collection being searched. Chap ter 4 describes a number of algorithms for constructing the inverted index from a text collection with particular attention to highly scalable and dis tributed algorithms that can be applied to very large collections. Chapter 5 covers techniques for compressing dictionaries and inverted indexes. These techniques are critical for achieving subsecond response times to user queries in large search engines. The indexes and queries considered in Chapters 1–5 only deal with *Boolean retrieval*, in which a document either matches a query, or does not. A desire to measure the *extent* to which a document matches a query, or the score of a document for a query, motivates the development of term weighting and the computation of scores in Chapters 6 and 7, leading to the idea of a list of documents that are rank-ordered for a query. Chapter 8 focuses on the evaluation of an information retrieval system based on the

Online edition (c)2009 Cambridge UP

*Preface xxxiii*

relevance of the documents it retrieves, allowing us to compare the relative performances of different systems on benchmark document collections and queries.

Chapters 9–21 build on the foundation of the first eight chapters to cover a variety of more advanced topics. Chapter 9 discusses methods by which retrieval can be enhanced through the use of techniques like relevance feed back and query expansion, which aim at increasing the likelihood of retriev ing relevant documents. Chapter 10 considers information retrieval from documents that are structured with markup languages like XML and HTML. We treat structured retrieval by reducing it to the vector space scoring meth ods developed in Chapter 6. Chapters 11 and 12 invoke probability theory to compute scores for documents on queries. Chapter 11 develops traditional probabilistic information retrieval, which provides a framework for comput ing the probability of relevance of a document, given a set of query terms. This probability may then be used as a score in ranking. Chapter 12 illus trates an alternative, wherein for each document in a collection, we build a language model from which one can estimate a probability that the language model generates a given query. This probability is another quantity with which we can rank-order documents.

Chapters 13–17 give a treatment of various forms of machine learning and numerical methods in information retrieval. Chapters 13–15 treat the prob lem of classifying documents into a set of known categories, given a set of documents along with the classes they belong to. Chapter 13 motivates sta tistical classification as one of the key technologies needed for a successful search engine, introduces Naive Bayes, a conceptually simple and efficient text classification method, and outlines the standard methodology for evalu ating text classifiers. Chapter 14 employs the vector space model from Chap ter 6 and introduces two classification methods, Rocchio and kNN, that op erate on document vectors. It also presents the bias-variance tradeoff as an important characterization of learning problems that provides criteria for se lecting an appropriate method for a text classification problem. Chapter 15 introduces support vector machines, which many researchers currently view as the most effective text classification method. We also develop connections in this chapter between the problem of classification and seemingly disparate topics such as the induction of scoring functions from a set of training exam ples.

Chapters 16–18 consider the problem of inducing clusters of related doc uments from a collection. In Chapter 16, we first give an overview of a number of important applications of clustering in information retrieval. We then describe two flat clustering algorithms: the *K*-means algorithm, an ef ficient and widely used document clustering method; and the Expectation Maximization algorithm, which is computationally more expensive, but also more flexible. Chapter 17 motivates the need for hierarchically structured

Online edition (c)2009 Cambridge UP

*xxxiv Preface*

clusterings (instead of flat clusterings) in many applications in information retrieval and introduces a number of clustering algorithms that produce a hierarchy of clusters. The chapter also addresses the difficult problem of automatically computing labels for clusters. Chapter 18 develops methods from linear algebra that constitute an extension of clustering, and also offer intriguing prospects for algebraic methods in information retrieval, which have been pursued in the approach of latent semantic indexing.

Chapters 19–21 treat the problem of web search. We give in Chapter 19 a

summary of the basic challenges in web search, together with a set of tech niques that are pervasive in web information retrieval. Next, Chapter 20 describes the architecture and requirements of a basic web crawler. Finally, Chapter 21 considers the power of link analysis in web search, using in the process several methods from linear algebra and advanced probability the ory.

This book is not comprehensive in covering all topics related to informa

tion retrieval. We have put aside a number of topics, which we deemed outside the scope of what we wished to cover in an introduction to infor mation retrieval class. Nevertheless, for people interested in these topics, we provide a few pointers to mainly textbook coverage here.

**Cross-language IR** (Grossman and Frieder 2004, ch. 4) and (Oard and Dorr 1996).

**Image and Multimedia IR** (Grossman and Frieder 2004, ch. 4), (Baeza-Yates and Ribeiro-Neto 1999, ch. 6), (Baeza-Yates and Ribeiro-Neto 1999, ch. 11),

(Baeza-Yates and Ribeiro-Neto 1999, ch. 12), (del Bimbo 1999), (Lew 2001),

and (Smeulders et al. 2000).

**Speech retrieval** (Coden et al. 2002).

**Music Retrieval** (Downie 2006) and http://www.ismir.net/.

**User interfaces for IR** (Baeza-Yates and Ribeiro-Neto 1999, ch. 10).

**Parallel and Peer-to-Peer IR** (Grossman and Frieder 2004, ch. 7), (Baeza-Yates and Ribeiro-Neto 1999, ch. 9), and (Aberer 2001).

**Digital libraries** (Baeza-Yates and Ribeiro-Neto 1999, ch. 15) and (Lesk 2004).

**Information science perspective** (Korfhage 1997), (Meadow et al. 1999), and (Ingwersen and Järvelin 2005).

**Logic-based approaches to IR** (van Rijsbergen 1989).

**Natural Language Processing techniques** (Manning and Schütze 1999), (Ju rafsky and Martin 2008), and (Lewis and Jones 1996).

Online edition (c)2009 Cambridge UP

*Preface xxxv*

**Prerequisites**

Introductory courses in data structures and algorithms, in linear algebra and in probability theory suffice as prerequisites for all 21 chapters. We now give more detail for the benefit of readers and instructors who wish to tailor their reading to some of the chapters.

Chapters 1–5 assume as prerequisite a basic course in algorithms and data structures. Chapters 6 and 7 require, in addition, a knowledge of basic lin ear algebra including vectors and dot products. No additional prerequisites are assumed until Chapter 11, where a basic course in probability theory is required; Section 11.1 gives a quick review of the concepts necessary in Chap ters 11–13. Chapter 15 assumes that the reader is familiar with the notion of nonlinear optimization, although the chapter may be read without detailed knowledge of algorithms for nonlinear optimization. Chapter 18 demands a first course in linear algebra including familiarity with the notions of matrix rank and eigenvectors; a brief review is given in Section 18.1. The knowledge of eigenvalues and eigenvectors is also necessary in Chapter 21.

**Book layout**

✎ Worked examples in the text appear with a pencil sign next to them in the left

margin. Advanced or difficult material appears in sections or subsections ✄ indicated with scissors in the margin. Exercises are marked in the margin

with a question mark. The level of difficulty of exercises is indicated as easy (⋆), medium (⋆⋆), or difficult (⋆ ⋆ ⋆). *?*

**Acknowledgments**

We would like to thank Cambridge University Press for allowing us to make the draft book available online, which facilitated much of the feedback we have received while writing the book. We also thank Lauren Cowles, who has been an outstanding editor, providing several rounds of comments on each chapter, on matters of style, organization, and coverage, as well as de tailed comments on the subject matter of the book. To the extent that we have achieved our goals in writing this book, she deserves an important part of the credit.

We are very grateful to the many people who have given us comments, suggestions, and corrections based on draft versions of this book. We thank for providing various corrections and comments: Cheryl Aasheim, Josh At tenberg, Daniel Beck, Luc Bélanger, Georg Buscher, Tom Breuel, Daniel Bur ckhardt, Fazli Can, Dinquan Chen, Stephen Clark, Ernest Davis, Pedro Domin

gos, Rodrigo Panchiniak Fernandes, Paolo Ferragina, Alex Fraser, Norbert

Online edition (c)2009 Cambridge UP

*xxxvi Preface*

Fuhr, Vignesh Ganapathy, Elmer Garduno, Xiubo Geng, David Gondek, Ser

gio Govoni, Corinna Habets, Ben Handy, Donna Harman, Benjamin Haskell,

Thomas Hühn, Deepak Jain, Ralf Jankowitsch, Dinakar Jayarajan, Vinay Kakade, Mei Kobayashi, Wessel Kraaij, Rick Lafleur, Florian Laws, Hang Li, David

Losada, David Mann, Ennio Masi, Sven Meyer zu Eissen, Alexander Murzaku,

Gonzalo Navarro, Frank McCown, Paul McNamee, Christoph Müller, Scott

Olsson, Tao Qin, Megha Raghavan, Michal Rosen-Zvi, Klaus Rothenhäusler,

Kenyu L. Runner, Alexander Salamanca, Grigory Sapunov, Evgeny Shad

chnev, Tobias Scheffer, Nico Schlaefer, Ian Soboroff, Benno Stein, Marcin

Sydow, Andrew Turner, Jason Utt, Huey Vo, Travis Wade, Mike Walsh, Changliang Wang, Renjing Wang, and Thomas Zeume.

Many people gave us detailed feedback on individual chapters, either at

our request or through their own initiative. For this, we’re particularly grate

ful to: James Allan, Omar Alonso, Ismail Sengor Altingovde, Vo Ngoc Anh,

Roi Blanco, Eric Breck, Eric Brown, Mark Carman, Carlos Castillo, Junghoo

Cho, Aron Culotta, Doug Cutting, Meghana Deodhar, Susan Dumais, Jo

hannes Fürnkranz, Andreas Heß, Djoerd Hiemstra, David Hull, Thorsten

Joachims, Siddharth Jonathan J. B., Jaap Kamps, Mounia Lalmas, Amy Langville, Nicholas Lester, Dave Lewis, Daniel Lowd, Yosi Mass, Jeff Michels, Alessan

dro Moschitti, Amir Najmi, Marc Najork, Giorgio Maria Di Nunzio, Paul

Ogilvie, Priyank Patel, Jan Pedersen, Kathryn Pedings, Vassilis Plachouras,

Daniel Ramage, Ghulam Raza, Stefan Riezler, Michael Schiehlen, Helmut

Schmid, Falk Nicolas Scholer, Sabine Schulte im Walde, Fabrizio Sebastiani,

Sarabjeet Singh, Valentin Spitkovsky, Alexander Strehl, John Tait, Shivaku

mar Vaithyanathan, Ellen Voorhees, Gerhard Weikum, Dawid Weiss, Yiming

Yang, Yisong Yue, Jian Zhang, and Justin Zobel.

And finally there were a few reviewers who absolutely stood out in terms

of the quality and quantity of comments that they provided. We thank them

for their significant impact on the content and structure of the book. We

express our gratitude to Pavel Berkhin, Stefan Büttcher, Jamie Callan, Byron

Dom, Torsten Suel, and Andrew Trotman.

Parts of the initial drafts of Chapters 13–15 were based on slides that were

generously provided by Ray Mooney. While the material has gone through

extensive revisions, we gratefully acknowledge Ray’s contribution to the

three chapters in general and to the description of the time complexities of

text classification algorithms in particular.

The above is unfortunately an incomplete list: we are still in the process of

incorporating feedback we have received. And, like all opinionated authors,

we did not always heed the advice that was so freely given. The published

versions of the chapters remain solely the responsibility of the authors.

The authors thank Stanford University and the University of Stuttgart for

providing a stimulating academic environment for discussing ideas and the

opportunity to teach courses from which this book arose and in which its

Online edition (c)2009 Cambridge UP

*Preface xxxvii*

contents were refined. CM thanks his family for the many hours they’ve let him spend working on this book, and hopes he’ll have a bit more free time on weekends next year. PR thanks his family for their patient support through the writing of this book and is also grateful to Yahoo! Inc. for providing a fertile environment in which to work on this book. HS would like to thank his parents, family, and friends for their support while writing this book.

**Web and contact information**

This book has a companion website at http://informationretrieval.org. As well as links to some more general resources, it is our intent to maintain on this web site a set of slides for each chapter which may be used for the corresponding lecture. We gladly welcome further feedback, corrections, and suggestions on the book, which may be sent to all the authors at informationretrieval (at) yahoogroups (dot) com.

Online edition (c)2009 Cambridge UP

DRAFT! © April 1, 2009 Cambridge University Press. Feedback welcome. *1* **1 *Boolean retrieval***

The meaning of the term *information retrieval* can be very broad. Just getting a credit card out of your wallet so that you can type in the card number is a form of information retrieval. However, as an academic field of study, INFORMATION *information retrieval* might be defined thus:

RETRIEVAL

Information retrieval (IR) is finding material (usually documents) of

an unstructured nature (usually text) that satisfies an information need

from within large collections (usually stored on computers).

As defined in this way, information retrieval used to be an activity that only a few people engaged in: reference librarians, paralegals, and similar pro fessional searchers. Now the world has changed, and hundreds of millions of people engage in information retrieval every day when they use a web search engine or search their email.1Information retrieval is fast becoming the dominant form of information access, overtaking traditional database style searching (the sort that is going on when a clerk says to you: “I’m sorry, I can only look up your order if you can give me your Order ID”).

IR can also cover other kinds of data and information problems beyond that specified in the core definition above. The term “unstructured data” refers to data which does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a relational database, of the sort companies usually use to main tain product inventories and personnel records. In reality, almost no data are truly “unstructured”. This is definitely true of all text data if you count the latent linguistic structure of human languages. But even accepting that the intended notion of structure is overt structure, most text has structure, such as headings and paragraphs and footnotes, which is commonly repre sented in documents by explicit markup (such as the coding underlying web

1. In modern parlance, the word “search” has tended to replace “(information) retrieval”; the term “search” is quite ambiguous, but in context we use the two synonymously.

Online edition (c)2009 Cambridge UP

*2 1 Boolean retrieval*

pages). IR is also used to facilitate “semistructured” search such as finding a document where the title contains Java and the body contains threading.

The field of information retrieval also covers supporting users in browsing

or filtering document collections or further processing a set of retrieved doc uments. Given a set of documents, clustering is the task of coming up with a good grouping of the documents based on their contents. It is similar to ar ranging books on a bookshelf according to their topic. Given a set of topics, standing information needs, or other categories (such as suitability of texts for different age groups), classification is the task of deciding which class(es), if any, each of a set of documents belongs to. It is often approached by first manually classifying some documents and then hoping to be able to classify new documents automatically.

Information retrieval systems can also be distinguished by the scale at

which they operate, and it is useful to distinguish three prominent scales. In *web search*, the system has to provide search over billions of documents stored on millions of computers. Distinctive issues are needing to gather documents for indexing, being able to build systems that work efficiently at this enormous scale, and handling particular aspects of the web, such as the exploitation of hypertext and not being fooled by site providers manip ulating page content in an attempt to boost their search engine rankings, given the commercial importance of the web. We focus on all these issues in Chapters 19–21. At the other extreme is *personal information retrieval*. In the last few years, consumer operating systems have integrated information retrieval (such as Apple’s Mac OS X Spotlight or Windows Vista’s Instant Search). Email programs usually not only provide search but also text clas sification: they at least provide a spam (junk mail) filter, and commonly also provide either manual or automatic means for classifying mail so that it can be placed directly into particular folders. Distinctive issues here include han dling the broad range of document types on a typical personal computer, and making the search system maintenance free and sufficiently lightweight in terms of startup, processing, and disk space usage that it can run on one machine without annoying its owner. In between is the space of *enterprise, institutional, and domain-specific search*, where retrieval might be provided for collections such as a corporation’s internal documents, a database of patents, or research articles on biochemistry. In this case, the documents will typi cally be stored on centralized file systems and one or a handful of dedicated machines will provide search over the collection. This book contains tech niques of value over this whole spectrum, but our coverage of some aspects of parallel and distributed search in web-scale search systems is compara tively light owing to the relatively small published literature on the details of such systems. However, outside of a handful of web search companies, a software developer is most likely to encounter the personal search and en terprise scenarios.

Online edition (c)2009 Cambridge UP

*1.1 An example information retrieval problem 3*

In this chapter we begin with a very simple example of an information retrieval problem, and introduce the idea of a term-document matrix (Sec tion 1.1) and the central inverted index data structure (Section 1.2). We will then examine the Boolean retrieval model and how Boolean queries are pro cessed (Sections 1.3 and 1.4).

**1.1 An example information retrieval problem**

A fat book which many people own is Shakespeare’s Collected Works. Sup pose you wanted to determine which plays of Shakespeare contain the words Brutus AND Caesar AND NOT Calpurnia. One way to do that is to start at the beginning and to read through all the text, noting for each play whether it contains Brutus and Caesar and excluding it from consideration if it con tains Calpurnia. The simplest form of document retrieval is for a computer to do this sort of linear scan through documents. This process is commonly

GREP referred to as *grepping* through text, after the Unix command grep, which performs this process. Grepping through text can be a very effective process, especially given the speed of modern computers, and often allows useful possibilities for wildcard pattern matching through the use of regular expres sions. With modern computers, for simple querying of modest collections (the size of Shakespeare’s Collected Works is a bit under one million words of text in total), you really need nothing more.

But for many purposes, you do need more:

1. To process large document collections quickly. The amount of online data has grown at least as quickly as the speed of computers, and we would now like to be able to search collections that total in the order of billions to trillions of words.

2. To allow more flexible matching operations. For example, it is impractical to perform the query Romans NEAR countrymen with grep, where NEAR might be defined as “within 5 words” or “within the same sentence”.

3. To allow ranked retrieval: in many cases you want the best answer to an information need among many documents that contain certain words.

INDEX The way to avoid linearly scanning the texts for each query is to *index* the documents in advance. Let us stick with Shakespeare’s Collected Works, and use it to introduce the basics of the Boolean retrieval model. Suppose we record for each document – here a play of Shakespeare’s – whether it contains each word out of all the words Shakespeare used (Shakespeare used

INCIDENCE MATRIX about 32,000 different words). The result is a binary term-document *incidence* TERM *matrix*, as in Figure 1.1. *Terms* are the indexed units (further discussed in Section 2.2); they are usually words, and for the moment you can think of

Online edition (c)2009 Cambridge UP

*4 1 Boolean retrieval*

Antony Julius The Hamlet Othello Macbeth . . .

and Caesar Tempest

Cleopatra

Antony 1 1 0 0 0 1

Brutus 1 1 0 1 0 0

Caesar 1 1 0 1 1 1

Calpurnia 0 1 0 0 0 0

Cleopatra 1 0 0 0 0 0

mercy 1 0 1 1 1 1

worser 1 0 1 1 1 0

. . .

◮ **Figure 1.1** A term-document incidence matrix. Matrix element (*t*, *d*) is 1 if the play in column *d* contains the word in row *t*, and is 0 otherwise.

them as words, but the information retrieval literature normally speaks of terms because some of them, such as perhaps I-9 or Hong Kong are not usually thought of as words. Now, depending on whether we look at the matrix rows or columns, we can have a vector for each term, which shows the documents it appears in, or a vector for each document, showing the terms that occur in it.2

To answer the query Brutus AND Caesar AND NOT Calpurnia, we take the

vectors for Brutus, Caesar and Calpurnia, complement the last, and then do a bitwise AND:

110100 AND 110111 AND 101111 = 100100

The answers for this query are thus *Antony and Cleopatra* and *Hamlet* (Fig ure 1.2).

BOOLEAN RETRIEVAL The *Boolean retrieval model* is a model for information retrieval in which we MODEL can pose any query which is in the form of a Boolean expression of terms, that is, in which terms are combined with the operators AND, OR, and NOT. The model views each document as just a set of words.

Let us now consider a more realistic scenario, simultaneously using the

opportunity to introduce some terminology and notation. Suppose we have DOCUMENT *N* = 1 million documents. By *documents* we mean whatever units we have decided to build a retrieval system over. They might be individual memos or chapters of a book (see Section 2.1.2 (page 20) for further discussion). We will refer to the group of documents over which we perform retrieval as the COLLECTION (document) *collection*. It is sometimes also referred to as a *corpus* (a *body* of CORPUS texts). Suppose each document is about 1000 words long (2–3 book pages). If

2. Formally, we take the transpose of the matrix to be able to get the terms as column vectors.

Online edition (c)2009 Cambridge UP

*1.1 An example information retrieval problem 5*

*Antony and Cleopatra, Act III, Scene ii*

Agrippa [Aside to Domitius Enobarbus]: Why, Enobarbus,

When Antony found Julius Caesar dead,

He cried almost to roaring; and he wept

When at Philippi he found Brutus slain.

*Hamlet, Act III, Scene ii*

Lord Polonius: I did enact Julius Caesar: I was killed i’ the

Capitol; Brutus killed me.

◮ **Figure 1.2** Results from Shakespeare for the query Brutus AND Caesar AND NOT Calpurnia.

we assume an average of 6 bytes per word including spaces and punctuation, then this is a document collection about 6 GB in size. Typically, there might be about *M* = 500,000 distinct terms in these documents. There is nothing special about the numbers we have chosen, and they might vary by an order of magnitude or more, but they give us some idea of the dimensions of the kinds of problems we need to handle. We will discuss and model these size assumptions in Section 5.1 (page 86).

AD HOC RETRIEVAL Our goal is to develop a system to address the *ad hoc retrieval* task. This is the most standard IR task. In it, a system aims to provide documents from within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query. An

INFORMATION NEED *information need* is the topic about which the user desires to know more, and QUERY is differentiated from a *query*, which is what the user conveys to the com puter in an attempt to communicate the information need. A document is RELEVANCE *relevant* if it is one that the user perceives as containing information of value with respect to their personal information need. Our example above was rather artificial in that the information need was defined in terms of par ticular words, whereas usually a user is interested in a topic like “pipeline leaks” and would like to find relevant documents regardless of whether they precisely use those words or express the concept with other words such as EFFECTIVENESS pipeline rupture. To assess the *effectiveness* of an IR system (i.e., the quality of its search results), a user will usually want to know two key statistics about the system’s returned results for a query:

PRECISION *Precision*: What fraction of the returned results are relevant to the informa tion need?

RECALL *Recall*: What fraction of the relevant documents in the collection were re turned by the system?

Online edition (c)2009 Cambridge UP

*6 1 Boolean retrieval*

Detailed discussion of relevance and evaluation measures including preci sion and recall is found in Chapter 8.

We now cannot build a term-document matrix in a naive way. A 500K ×

1M matrix has half-a-trillion 0’s and 1’s – too many to fit in a computer’s memory. But the crucial observation is that the matrix is extremely sparse, that is, it has few non-zero entries. Because each document is 1000 words long, the matrix has no more than one billion 1’s, so a minimum of 99.8% of the cells are zero. A much better representation is to record only the things that do occur, that is, the 1 positions.

This idea is central to the first major concept in information retrieval, the

INVERTED INDEX *inverted index*. The name is actually redundant: an index always maps back from terms to the parts of a document where they occur. Nevertheless, *in verted index*, or sometimes *inverted file*, has become the standard term in infor mation retrieval.3 The basic idea of an inverted index is shown in Figure 1.3.

DICTIONARY We keep a *dictionary* of terms (sometimes also referred to as a *vocabulary* or VOCABULARY *lexicon*; in this book, we use *dictionary* for the data structure and *vocabulary* LEXICON for the set of terms). Then for each term, we have a list that records which documents the term occurs in. Each item in the list – which records that a term appeared in a document (and, later, often, the positions in the docu ment) – is conventionally called a *posting*.4

POSTING The list is then called a *postings* POSTINGS LIST *list* (or inverted list), and all the postings lists taken together are referred to as POSTINGS the *postings*. The dictionary in Figure 1.3 has been sorted alphabetically and each postings list is sorted by document ID. We will see why this is useful in Section 1.3, below, but later we will also consider alternatives to doing this (Section 7.1.5).

**1.2 A first take at building an inverted index**

To gain the speed benefits of indexing at retrieval time, we have to build the index in advance. The major steps in this are:

1. Collect the documents to be indexed:

Friends, Romans, countrymen. So let it be with Caesar . . .

2. Tokenize the text, turning each document into a list of tokens:

Friends Romans countrymen So . . .

3. Some information retrieval researchers prefer the term inverted file, but expressions like in dex construction and index compression are much more common than inverted file construction and inverted file compression. For consistency, we use (inverted) index throughout this book.

4. In a (non-positional) inverted index, a posting is just a document ID, but it is inherently associated with a term, via the postings list it is placed on; sometimes we will also talk of a (term, docID) pair as a posting.

Online edition (c)2009 Cambridge UP

*1.2 A first take at building an inverted index 7*

Brutus −→ 1 2 4 11 31 45 173 174

Caesar −→ 1 2 4 5 6 16 57 132 . . .

Calpurnia −→ 2 31 54 101

...

| {z } | {z }

**Dictionary Postings**

◮ **Figure 1.3** The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms: friend roman countryman so . . .

4. Index the documents that each term occurs in by creating an inverted in dex, consisting of a dictionary and postings.

We will define and discuss the earlier stages of processing, that is, steps 1–3, in Section 2.2 (page 22). Until then you can think of *tokens* and *normalized tokens* as also loosely equivalent to *words*. Here, we assume that the first 3 steps have already been done, and we examine building a basic inverted index by sort-based indexing.

Within a document collection, we assume that each document has a unique DOCID serial number, known as the document identifier (*docID*). During index con struction, we can simply assign successive integers to each new document when it is first encountered. The input to indexing is a list of normalized tokens for each document, which we can equally think of as a list of pairs of SORTING term and docID, as in Figure 1.4. The core indexing step is *sorting* this list so that the terms are alphabetical, giving us the representation in the middle column of Figure 1.4. Multiple occurrences of the same term from the same document are then merged.5Instances of the same term are then grouped, and the result is split into a *dictionary* and *postings*, as shown in the right column of Figure 1.4. Since a term generally occurs in a number of docu ments, this data organization already reduces the storage requirements of the index. The dictionary also records some statistics, such as the number of DOCUMENT documents which contain each term (the *document frequency*, which is here FREQUENCY also the length of each postings list). This information is not vital for a ba sic Boolean search engine, but it allows us to improve the efficiency of the

5. Unix users can note that these steps are similar to use of the sort and then uniq commands.

Online edition (c)2009 Cambridge UP

*8 1 Boolean retrieval* **Doc 1 Doc 2**

I did enact Julius Caesar: I was killed i’ the Capitol; Brutus killed me.

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

**term docID** I 1 did 1 enact 1 julius 1 caesar 1 I 1 was 1 killed 1 i’ 1 the 1 capitol 1 brutus 1 killed 1 me 1 so 2 let 2 it 2 be 2 with 2 caesar 2 the 2 noble 2 brutus 2 hath 2 told 2 you 2 caesar 2 was 2 ambitious 2

=⇒

**term docID** ambitious 2 be 2 brutus 1 brutus 2 capitol 1 caesar 1 caesar 2 caesar 2 did 1 enact 1 hath 1 I 1 I 1 i’ 1 it 2 julius 1 killed 1 killed 1 let 2 me 1 noble 2 so 2 the 1 the 2 told 2 you 2 was 1 was 2 with 2

=⇒

**term doc. freq.** → **postings lists** ambitious 1 → 2

be 1 → 2

brutus 2 → 1 → 2 capitol 1 → 1

caesar 2 → 1 → 2 did 1 → 1

enact 1 → 1

hath 1 → 2

I 1 → 1

i’ 1 → 1

it 1 → 2

julius 1 → 1

killed 1 → 1

let 1 → 2

me 1 → 1

noble 1 → 2

so 1 → 2

the 2 → 1 → 2 told 1 → 2

you 1 → 2

was 2 → 1 → 2 with 1 → 2

◮ **Figure 1.4** Building an index by sorting and grouping. The sequence of terms in each document, tagged by their documentID (left) is sorted alphabetically (mid dle). Instances of the same term are then grouped by word and then by documentID. The terms and documentIDs are then separated out (right). The dictionary stores the terms, and has a pointer to the postings list for each term. It commonly also stores other summary information such as, here, the document frequency of each term. We use this information for improving query time efficiency and, later, for weighting in ranked retrieval models. Each postings list stores the list of documents in which a term occurs, and may store other information such as the term frequency (the frequency of each term in each document) or the position(s) of the term in each document.

Online edition (c)2009 Cambridge UP

*1.2 A first take at building an inverted index 9*

search engine at query time, and it is a statistic later used in many ranked re trieval models. The postings are secondarily sorted by docID. This provides the basis for efficient query processing. This inverted index structure is es sentially without rivals as the most efficient structure for supporting ad hoc text search.

In the resulting index, we pay for storage of both the dictionary and the postings lists. The latter are much larger, but the dictionary is commonly kept in memory, while postings lists are normally kept on disk, so the size of each is important, and in Chapter 5 we will examine how each can be optimized for storage and access efficiency. What data structure should be used for a postings list? A fixed length array would be wasteful as some words occur in many documents, and others in very few. For an in-memory postings list, two good alternatives are singly linked lists or variable length arrays. Singly linked lists allow cheap insertion of documents into postings lists (following updates, such as when recrawling the web for updated doc uments), and naturally extend to more advanced indexing strategies such as skip lists (Section 2.3), which require additional pointers. Variable length ar rays win in space requirements by avoiding the overhead for pointers and in time requirements because their use of contiguous memory increases speed on modern processors with memory caches. Extra pointers can in practice be encoded into the lists as offsets. If updates are relatively infrequent, variable length arrays will be more compact and faster to traverse. We can also use a hybrid scheme with a linked list of fixed length arrays for each term. When postings lists are stored on disk, they are stored (perhaps compressed) as a contiguous run of postings without explicit pointers (as in Figure 1.3), so as to minimize the size of the postings list and the number of disk seeks to read a postings list into memory.

*?***Exercise 1.1** [⋆]

Draw the inverted index that would be built for the following document collection. (See Figure 1.3 for an example.)

**Doc 1** new home sales top forecasts

**Doc 2** home sales rise in july

**Doc 3** increase in home sales in july

**Doc 4** july new home sales rise

**Exercise 1.2** [⋆] Consider these documents:

**Doc 1** breakthrough drug for schizophrenia

**Doc 2** new schizophrenia drug

**Doc 3** new approach for treatment of schizophrenia

**Doc 4** new hopes for schizophrenia patients

a. Draw the term-document incidence matrix for this document collection.

Online edition (c)2009 Cambridge UP

*10 1 Boolean retrieval*

Brutus −→ 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174

Calpurnia −→ 2 → 31 → 54 → 101

Intersection =⇒ 2 → 31

◮ **Figure 1.5** Intersecting the postings lists for Brutus and Calpurnia from Figure 1.3.

b. Draw the inverted index representation for this collection, as in Figure 1.3 (page 7).

**Exercise 1.3** [⋆]

For the document collection shown in Exercise 1.2, what are the returned results for these queries:

a. schizophrenia AND drug

b. for AND NOT(drug OR approach)

**1.3 Processing Boolean queries**

How do we process a query using an inverted index and the basic Boolean SIMPLE CONJUNCTIVE retrieval model? Consider processing the *simple conjunctive query*:

QUERIES

(1.1) Brutus AND Calpurnia

over the inverted index partially shown in Figure 1.3 (page 7). We:

1. Locate Brutus in the Dictionary

2. Retrieve its postings

3. Locate Calpurnia in the Dictionary

4. Retrieve its postings

5. Intersect the two postings lists, as shown in Figure 1.5.

POSTINGS LIST The *intersection* operation is the crucial one: we need to efficiently intersect INTERSECTION postings lists so as to be able to quickly find documents that contain both POSTINGS MERGE terms. (This operation is sometimes referred to as *merging* postings lists: this slightly counterintuitive name reflects using the term *merge algorithm* for a general family of algorithms that combine multiple sorted lists by inter leaved advancing of pointers through each; here we are merging the lists with a logical AND operation.)

There is a simple and effective method of intersecting postings lists using

the merge algorithm (see Figure 1.6): we maintain pointers into both lists

Online edition (c)2009 Cambridge UP

*1.3 Processing Boolean queries 11*

INTERSECT(*p*1, *p*2)

1 *answer* ← h i

2 **while** *p*1 6= NIL and *p*2 6= NIL

3 **do if** *docID*(*p*1) = *docID*(*p*2)

4 **then** ADD(*answer*, *docID*(*p*1))

5 *p*1 ← *next*(*p*1)

6 *p*2 ← *next*(*p*2)

7 **else if** *docID*(*p*1) < *docID*(*p*2)

8 **then** *p*1 ← *next*(*p*1)

9 **else** *p*2 ← *next*(*p*2)

10 **return** *answer*

◮ **Figure 1.6** Algorithm for the intersection of two postings lists *p*1 and *p*2.

and walk through the two postings lists simultaneously, in time linear in the total number of postings entries. At each step, we compare the docID pointed to by both pointers. If they are the same, we put that docID in the results list, and advance both pointers. Otherwise we advance the pointer pointing to the smaller docID. If the lengths of the postings lists are *x* and *y*, the intersection takes *O*(*x* + *y*) operations. Formally, the complexity of querying is Θ(*N*), where *N* is the number of documents in the collection.6 Our indexing methods gain us just a constant, not a difference in Θ time complexity compared to a linear scan, but in practice the constant is huge. To use this algorithm, it is crucial that postings be sorted by a single global ordering. Using a numeric sort by docID is one simple way to achieve this.

We can extend the intersection operation to process more complicated queries like:

(1.2) (Brutus OR Caesar) AND NOT Calpurnia

QUERY OPTIMIZATION *Query optimization* is the process of selecting how to organize the work of an swering a query so that the least total amount of work needs to be done by the system. A major element of this for Boolean queries is the order in which postings lists are accessed. What is the best order for query processing? Con sider a query that is an AND of *t* terms, for instance:

(1.3) Brutus AND Caesar AND Calpurnia

For each of the *t* terms, we need to get its postings, then AND them together. The standard heuristic is to process terms in order of increasing document

6. The notation Θ(·) is used to express an asymptotically tight bound on the complexity of an algorithm. Informally, this is often written as *O*(·), but this notation really expresses an asymptotic upper bound, which need not be tight (Cormen et al. 1990).

Online edition (c)2009 Cambridge UP

*12 1 Boolean retrieval*

INTERSECT(h*t*1, . . . , *tn*i)

1 *terms* ← SORTBYINCREASINGFREQUENCY(h*t*1, . . . , *tn*i)

2 *result* ← *postings*(*f irst*(*terms*))

3 *terms* ← *rest*(*terms*)

4 **while** *terms* 6= NIL and *result* 6= NIL

5 **do** *result* ← INTERSECT(*result*, *postings*(*f irst*(*terms*)))

6 *terms* ← *rest*(*terms*)

7 **return** *result*

◮ **Figure 1.7** Algorithm for conjunctive queries that returns the set of documents containing each term in the input list of terms.

frequency: if we start by intersecting the two smallest postings lists, then all intermediate results must be no bigger than the smallest postings list, and we are therefore likely to do the least amount of total work. So, for the postings lists in Figure 1.3 (page 7), we execute the above query as:

(1.4) (Calpurnia AND Brutus) AND Caesar

This is a first justification for keeping the frequency of terms in the dictionary: it allows us to make this ordering decision based on in-memory data before accessing any postings list.

Consider now the optimization of more general queries, such as:

(1.5) (madding OR crowd) AND (ignoble OR strife) AND (killed OR slain)

As before, we will get the frequencies for all terms, and we can then (con servatively) estimate the size of each OR by the sum of the frequencies of its disjuncts. We can then process the query in increasing order of the size of each disjunctive term.

For arbitrary Boolean queries, we have to evaluate and temporarily store

the answers for intermediate expressions in a complex expression. However, in many circumstances, either because of the nature of the query language, or just because this is the most common type of query that users submit, a query is purely conjunctive. In this case, rather than viewing merging post ings lists as a function with two inputs and a distinct output, it is more ef ficient to intersect each retrieved postings list with the current intermediate result in memory, where we initialize the intermediate result by loading the postings list of the least frequent term. This algorithm is shown in Figure 1.7. The intersection operation is then asymmetric: the intermediate results list is in memory while the list it is being intersected with is being read from disk. Moreover the intermediate results list is always at least as short as the other list, and in many cases it is orders of magnitude shorter. The postings

Online edition (c)2009 Cambridge UP

*1.3 Processing Boolean queries 13*

intersection can still be done by the algorithm in Figure 1.6, but when the difference between the list lengths is very large, opportunities to use alter native techniques open up. The intersection can be calculated in place by destructively modifying or marking invalid items in the intermediate results list. Or the intersection can be done as a sequence of binary searches in the long postings lists for each posting in the intermediate results list. Another possibility is to store the long postings list as a hashtable, so that membership of an intermediate result item can be calculated in constant rather than linear or log time. However, such alternative techniques are difficult to combine with postings list compression of the sort discussed in Chapter 5. Moreover, standard postings list intersection operations remain necessary when both terms of a query are very common.

*?***Exercise 1.4** [⋆]

For the queries below, can we still run through the intersection in time *O*(*x* + *y*), where *x* and *y* are the lengths of the postings lists for Brutus and Caesar? If not, what can we achieve?

a. Brutus AND NOT Caesar

b. Brutus OR NOT Caesar

**Exercise 1.5** [⋆]

Extend the postings merge algorithm to arbitrary Boolean query formulas. What is its time complexity? For instance, consider:

c. (Brutus OR Caesar) AND NOT (Antony OR Cleopatra)

Can we always merge in linear time? Linear in what? Can we do better than this?

**Exercise 1.6** [⋆⋆] We can use distributive laws for AND and OR to rewrite queries.

a. Show how to rewrite the query in Exercise 1.5 into disjunctive normal form using the distributive laws.

b. Would the resulting query be more or less efficiently evaluated than the original form of this query?

c. Is this result true in general or does it depend on the words and the contents of the document collection?

**Exercise 1.7** [⋆] Recommend a query processing order for

d. (tangerine OR trees) AND (marmalade OR skies) AND (kaleidoscope OR eyes) given the following postings list sizes:

Online edition (c)2009 Cambridge UP

*14 1 Boolean retrieval*

**Term Postings size**

eyes 213312

kaleidoscope 87009

marmalade 107913

skies 271658

tangerine 46653

trees 316812

**Exercise 1.8** [⋆] If the query is:

e. friends AND romans AND (NOT countrymen)

how could we use the frequency of countrymen in evaluating the best query evaluation order? In particular, propose a way of handling negation in determining the order of query processing.

**Exercise 1.9** [⋆⋆]

For a conjunctive query, is processing postings lists in order of size guaranteed to be optimal? Explain why it is, or give an example where it isn’t.

**Exercise 1.10** [⋆⋆]

Write out a postings merge algorithm, in the style of Figure 1.6 (page 11), for an *x* OR *y* query.

**Exercise 1.11** [⋆⋆]

How should the Boolean query *x* AND NOT *y* be handled? Why is naive evaluation of this query normally very expensive? Write out a postings merge algorithm that evaluates this query efficiently.

**1.4 The extended Boolean model versus ranked retrieval**

RANKED RETRIEVAL The Boolean retrieval model contrasts with *ranked retrieval models* such as the MODEL vector space model (Section 6.3), in which users largely use *free text queries*, FREE TEXT QUERIES that is, just typing one or more words rather than using a precise language with operators for building up query expressions, and the system decides which documents best satisfy the query. Despite decades of academic re search on the advantages of ranked retrieval, systems implementing the Boo lean retrieval model were the main or only search option provided by large commercial information providers for three decades until the early 1990s (ap proximately the date of arrival of the World Wide Web). However, these systems did not have just the basic Boolean operations (AND, OR, and NOT) which we have presented so far. A strict Boolean expression over terms with an unordered results set is too limited for many of the information needs that people have, and these systems implemented extended Boolean retrieval models by incorporating additional operators such as term proximity oper PROXIMITY OPERATOR ators. A *proximity operator* is a way of specifying that two terms in a query

Online edition (c)2009 Cambridge UP

*1.4 The extended Boolean model versus ranked retrieval 15*

must occur close to each other in a document, where closeness may be mea sured by limiting the allowed number of intervening words or by reference to a structural unit such as a sentence or paragraph.

✎ **Example 1.1: Commercial Boolean searching: Westlaw.** Westlaw (http://www.westlaw.com/)

is the largest commercial legal search service (in terms of the number of paying sub scribers), with over half a million subscribers performing millions of searches a day over tens of terabytes of text data. The service was started in 1975. In 2005, Boolean search (called “Terms and Connectors” by Westlaw) was still the default, and used by a large percentage of users, although ranked free text querying (called “Natural Language” by Westlaw) was added in 1992. Here are some example Boolean queries on Westlaw:

*Information need:* Information on the legal theories involved in preventing the

disclosure of trade secrets by employees formerly employed by a competing

company. *Query:* "trade secret" /s disclos! /s prevent /s employe!

*Information need:* Requirements for disabled people to be able to access a work

place.

*Query:* disab! /p access! /s work-site work-place (employment /3 place)

*Information need:* Cases about a host’s responsibility for drunk guests.

*Query:* host! /p (responsib! liab!) /p (intoxicat! drunk!) /p guest

Note the long, precise queries and the use of proximity operators, both uncommon in web search. Submitted queries average about ten words in length. Unlike web search conventions, a space between words represents disjunction (the tightest bind ing operator), & is AND and /s, /p, and /*k* ask for matches in the same sentence, same paragraph or within *k* words respectively. Double quotes give a *phrase search* (consecutive words); see Section 2.4 (page 39). The exclamation mark (!) gives a trail ing wildcard query (see Section 3.2, page 51); thus liab! matches all words starting with liab. Additionally work-site matches any of *worksite*, *work-site* or *work site*; see Section 2.2.1 (page 22). Typical expert queries are usually carefully defined and incre mentally developed until they obtain what look to be good results to the user.

Many users, particularly professionals, prefer Boolean query models. Boolean queries are precise: a document either matches the query or it does not. This of fers the user greater control and transparency over what is retrieved. And some do mains, such as legal materials, allow an effective means of document ranking within a Boolean model: Westlaw returns documents in reverse chronological order, which is in practice quite effective. In 2007, the majority of law librarians still seem to rec ommend terms and connectors for high recall searches, and the majority of legal users think they are getting greater control by using them. However, this does not mean that Boolean queries are more effective for professional searchers. Indeed, ex perimenting on a Westlaw subcollection, Turtle (1994) found that free text queries produced better results than Boolean queries prepared by Westlaw’s own reference librarians for the majority of the information needs in his experiments. A general problem with Boolean search is that using AND operators tends to produce high pre cision but low recall searches, while using OR operators gives low precision but high recall searches, and it is difficult or impossible to find a satisfactory middle ground.

In this chapter, we have looked at the structure and construction of a basic

Online edition (c)2009 Cambridge UP

*16 1 Boolean retrieval*

inverted index, comprising a dictionary and postings lists. We introduced the Boolean retrieval model, and examined how to do efficient retrieval via linear time merges and simple query optimization. In Chapters 2–7 we will consider in detail richer query models and the sort of augmented index struc tures that are needed to handle them efficiently. Here we just mention a few of the main additional things we would like to be able to do:

1. We would like to better determine the set of terms in the dictionary and to provide retrieval that is tolerant to spelling mistakes and inconsistent

choice of words.

2. It is often useful to search for compounds or phrases that denote a concept such as “operating system”. As the Westlaw examples show, we might also

wish to do proximity queries such as Gates NEAR Microsoft. To answer

such queries, the index has to be augmented to capture the proximities of

terms in documents.

3. A Boolean model only records term presence or absence, but often we would like to accumulate evidence, giving more weight to documents that

have a term several times as opposed to ones that contain it only once. To

TERM FREQUENCY be able to do this we need *term frequency* information (the number of times a term occurs in a document) in postings lists.

4. Boolean queries just retrieve a set of matching documents, but commonly we wish to have an effective method to order (or “rank”) the returned

results. This requires having a mechanism for determining a document

score which encapsulates how good a match a document is for a query.

With these additional ideas, we will have seen most of the basic technol

ogy that supports ad hoc searching over unstructured information. Ad hoc searching over documents has recently conquered the world, powering not only web search engines but the kind of unstructured search that lies behind the large eCommerce websites. Although the main web search engines differ by emphasizing free text querying, most of the basic issues and technologies of indexing and querying remain the same, as we will see in later chapters. Moreover, over time, web search engines have added at least partial imple mentations of some of the most popular operators from extended Boolean models: phrase search is especially popular and most have a very partial implementation of Boolean operators. Nevertheless, while these options are liked by expert searchers, they are little used by most people and are not the main focus in work on trying to improve web search engine performance.

*?***Exercise 1.12** [⋆]

Write a query using Westlaw syntax which would find any of the words professor, teacher, or lecturer in the same sentence as a form of the verb explain.

Online edition (c)2009 Cambridge UP

*1.5 References and further reading 17*

**Exercise 1.13** [⋆]

Try using the Boolean search features on a couple of major web search engines. For instance, choose a word, such as burglar, and submit the queries (i) burglar, (ii) burglar AND burglar, and (iii) burglar OR burglar. Look at the estimated number of results and top hits. Do they make sense in terms of Boolean logic? Often they haven’t for major search engines. Can you make sense of what is going on? What about if you try different words? For example, query for (i) knight, (ii) conquer, and then (iii) knight OR conquer. What bound should the number of results from the first two queries place on the third query? Is this bound observed?

**1.5 References and further reading**

The practical pursuit of computerized information retrieval began in the late 1940s (Cleverdon 1991, Liddy 2005). A great increase in the production of scientific literature, much in the form of less formal technical reports rather than traditional journal articles, coupled with the availability of computers, led to interest in automatic document retrieval. However, in those days, doc ument retrieval was always based on author, title, and keywords; full-text search came much later.

The article of Bush (1945) provided lasting inspiration for the new field:

“Consider a future device for individual use, which is a sort of mech anized private file and library. It needs a name, and, to coin one at random, ‘memex’ will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mech anized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory.”

The term *Information Retrieval* was coined by Calvin Mooers in 1948/1950 (Mooers 1950).

In 1958, much newspaper attention was paid to demonstrations at a con ference (see Taube and Wooster 1958) of IBM “auto-indexing” machines, based primarily on the work of H. P. Luhn. Commercial interest quickly gravitated towards Boolean retrieval systems, but the early years saw a heady debate over various disparate technologies for retrieval systems. For example Moo ers (1961) dissented:

“It is a common fallacy, underwritten at this date by the investment of several million dollars in a variety of retrieval hardware, that the al gebra of George Boole (1847) is the appropriate formalism for retrieval system design. This view is as widely and uncritically accepted as it is wrong.”

The observation of AND vs. OR giving you opposite extremes in a precision/ recall tradeoff, but not the middle ground comes from (Lee and Fox 1988).

Online edition (c)2009 Cambridge UP

*18 1 Boolean retrieval*

The book (Witten et al. 1999) is the standard reference for an in-depth com

parison of the space and time efficiency of the inverted index versus other possible data structures; a more succinct and up-to-date presentation ap pears in Zobel and Moffat (2006). We further discuss several approaches in Chapter 5.

REGULAR EXPRESSIONS Friedl (2006) covers the practical usage of *regular expressions* for searching. The underlying computer science appears in (Hopcroft et al. 2000).

Online edition (c)2009 Cambridge UP

DRAFT! © April 1, 2009 Cambridge University Press. Feedback welcome. *19*

**2*The term vocabulary and postings lists***

Recall the major steps in inverted index construction:

1. Collect the documents to be indexed.

2. Tokenize the text.

3. Do linguistic preprocessing of tokens.

4. Index the documents that each term occurs in.

In this chapter we first briefly mention how the basic unit of a document can be defined and how the character sequence that it comprises is determined (Section 2.1). We then examine in detail some of the substantive linguis tic issues of tokenization and linguistic preprocessing, which determine the vocabulary of terms which a system uses (Section 2.2). Tokenization is the process of chopping character streams into tokens, while linguistic prepro cessing then deals with building equivalence classes of tokens which are the set of terms that are indexed. Indexing itself is covered in Chapters 1 and 4. Then we return to the implementation of postings lists. In Section 2.3, we examine an extended postings list data structure that supports faster query ing, while Section 2.4 covers building postings data structures suitable for handling phrase and proximity queries, of the sort that commonly appear in both extended Boolean models and on the web.

**2.1 Document delineation and character sequence decoding**

**2.1.1 Obtaining the character sequence in a document**

Digital documents that are the input to an indexing process are typically bytes in a file or on a web server. The first step of processing is to convert this byte sequence into a linear sequence of characters. For the case of plain En glish text in ASCII encoding, this is trivial. But often things get much more

Online edition (c)2009 Cambridge UP

*20 2 The term vocabulary and postings lists*

complex. The sequence of characters may be encoded by one of various sin gle byte or multibyte encoding schemes, such as Unicode UTF-8, or various national or vendor-specific standards. We need to determine the correct en coding. This can be regarded as a machine learning classification problem, as discussed in Chapter 13,1 but is often handled by heuristic methods, user selection, or by using provided document metadata. Once the encoding is determined, we decode the byte sequence to a character sequence. We might save the choice of encoding because it gives some evidence about what lan guage the document is written in.

The characters may have to be decoded out of some binary representation

like Microsoft Word DOC files and/or a compressed format such as zip files. Again, we must determine the document format, and then an appropriate decoder has to be used. Even for plain text documents, additional decoding may need to be done. In XML documents (Section 10.1, page 197), charac ter entities, such as &amp;, need to be decoded to give the correct character, namely & for &amp;. Finally, the textual part of the document may need to be extracted out of other material that will not be processed. This might be the desired handling for XML files, if the markup is going to be ignored; we would almost certainly want to do this with postscript or PDF files. We will not deal further with these issues in this book, and will assume henceforth that our documents are a list of characters. Commercial products usually need to support a broad range of document types and encodings, since users want things to just work with their data as is. Often, they just think of docu ments as text inside applications and are not even aware of how it is encoded on disk. This problem is usually solved by licensing a software library that handles decoding document formats and character encodings.

The idea that text is a linear sequence of characters is also called into ques

tion by some writing systems, such as Arabic, where text takes on some two dimensional and mixed order characteristics, as shown in Figures 2.1 and 2.2. But, despite some complicated writing system conventions, there is an underlying sequence of sounds being represented and hence an essen tially linear structure remains, and this is what is represented in the digital representation of Arabic, as shown in Figure 2.1.

**2.1.2 Choosing a document unit**

DOCUMENT UNIT The next phase is to determine what the *document unit* for indexing is. Thus far we have assumed that documents are fixed units for the purposes of in dexing. For example, we take each file in a folder as a document. But there

1. A classifier is a function that takes objects of some sort and assigns them to one of a number of distinct classes (see Chapter 13). Usually classification is done by machine learning methods such as probabilistic models, but it can also be done by hand-written rules.

Online edition (c)2009 Cambridge UP

*2.1 Document delineation and character sequence decoding 21*

ك ِ ت ا ب ٌ ⇐ آِ َ بٌ

un b ā t i k

/kitābun/ ‘a book’

◮ **Figure 2.1** An example of a vocalized Modern Standard Arabic word. The writing is from right to left and letters undergo complex mutations as they are combined. The representation of short vowels (here, /i/ and /u/) and the final /n/ (nunation) de parts from strict linearity by being represented as diacritics above and below letters. Nevertheless, the represented text is still clearly a linear ordering of characters repre senting sounds. Full vocalization, as here, normally appears only in the Koran and children’s books. Day-to-day text is unvocalized (short vowels are not represented but the letter for a would still appear) or partially vocalized, with short vow ¯ els in serted in places where the writer perceives ambiguities. These choices add further complexities to indexing.

ا 1962 132 ! "!"# ا ل ا .

ا

ا

← → ← → ← START

‘Algeria achieved its independence in 1962 after 132 years of French occupation.’

◮ **Figure 2.2** The conceptual linear order of characters is not necessarily the order that you see on the page. In languages that are written right-to-left, such as Hebrew and Arabic, it is quite common to also have left-to-right text interspersed, such as numbers and dollar amounts. With modern Unicode representation concepts, the order of characters in files matches the conceptual order, and the reversal of displayed characters is handled by the rendering system, but this may not be true for documents in older encodings.

are many cases in which you might want to do something different. A tra ditional Unix (mbox-format) email file stores a sequence of email messages (an email folder) in one file, but you might wish to regard each email mes sage as a separate document. Many email messages now contain attached documents, and you might then want to regard the email message and each contained attachment as separate documents. If an email message has an attached zip file, you might want to decode the zip file and regard each file it contains as a separate document. Going in the opposite direction, various pieces of web software (such as latex2html) take things that you might regard as a single document (e.g., a Powerpoint file or a LATEX document) and split them into separate HTML pages for each slide or subsection, stored as sep arate files. In these cases, you might want to combine multiple files into a single document.

INDEXING More generally, for very long documents, the issue of indexing *granularity* GRANULARITY arises. For a collection of books, it would usually be a bad idea to index an

Online edition (c)2009 Cambridge UP

*22 2 The term vocabulary and postings lists*

entire book as a document. A search for Chinese toys might bring up a book that mentions China in the first chapter and toys in the last chapter, but this does not make it relevant to the query. Instead, we may well wish to index each chapter or paragraph as a mini-document. Matches are then more likely to be relevant, and since the documents are smaller it will be much easier for the user to find the relevant passages in the document. But why stop there? We could treat individual sentences as mini-documents. It becomes clear that there is a precision/recall tradeoff here. If the units get too small, we are likely to miss important passages because terms were distributed over several mini-documents, while if units are too large we tend to get spurious matches and the relevant information is hard for the user to find.

The problems with large document units can be alleviated by use of ex

plicit or implicit proximity search (Sections 2.4.2 and 7.2.2), and the trade offs in resulting system performance that we are hinting at are discussed in Chapter 8. The issue of index granularity, and in particular a need to simultaneously index documents at multiple levels of granularity, appears prominently in XML retrieval, and is taken up again in Chapter 10. An IR system should be designed to offer choices of granularity. For this choice to be made well, the person who is deploying the system must have a good understanding of the document collection, the users, and their likely infor mation needs and usage patterns. For now, we will henceforth assume that a suitable size document unit has been chosen, together with an appropriate way of dividing or aggregating files, if needed.

**2.2 Determining the vocabulary of terms**

**2.2.1 Tokenization**

Given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called *tokens*, perhaps at the same time throwing away certain characters, such as punctuation. Here is an example of tokenization:

Input: Friends, Romans, Countrymen, lend me your ears;

Output: Friends Romans Countrymen lend me your ears

These tokens are often loosely referred to as terms or words, but it is some TOKEN times important to make a type/token distinction. A *token* is an instance of a sequence of characters in some particular document that are grouped TYPE together as a useful semantic unit for processing. A *type* is the class of all TERM tokens containing the same character sequence. A *term* is a (perhaps nor malized) type that is included in the IR system’s dictionary. The set of index terms could be entirely distinct from the tokens, for instance, they could be

Online edition (c)2009 Cambridge UP

*2.2 Determining the vocabulary of terms 23*

semantic identifiers in a taxonomy, but in practice in modern IR systems they are strongly related to the tokens in the document. However, rather than be ing exactly the tokens that appear in the document, they are usually derived from them by various normalization processes which are discussed in Sec tion 2.2.3.2 For example, if the document to be indexed is *to sleep perchance to dream*, then there are 5 tokens, but only 4 types (since there are 2 instances of *to*). However, if *to* is omitted from the index (as a stop word, see Section 2.2.2 (page 27)), then there will be only 3 terms: *sleep*, *perchance*, and *dream*.

The major question of the tokenization phase is what are the correct tokens to use? In this example, it looks fairly trivial: you chop on whitespace and throw away punctuation characters. This is a starting point, but even for English there are a number of tricky cases. For example, what do you do about the various uses of the apostrophe for possession and contractions?

Mr. O’Neill thinks that the boys’ stories about Chile’s capital aren’t amusing.

For *O’Neill*, which of the following is the desired tokenization?

neill

oneill

o’neill

o’ neill

o neill ?

And for *aren’t*, is it:

aren’t

arent

are n’t

aren t ?

A simple strategy is to just split on all non-alphanumeric characters, but while o neill looks okay, aren t looks intuitively bad. For all of them, the choices determine which Boolean queries will match. A query of neill AND capital will match in three cases but not the other two. In how many cases would a query of o’neill AND capital match? If no preprocessing of a query is done, then it would match in only one of the five cases. For either

2. That is, as defined here, tokens that are not indexed (stop words) are not terms, and if mul tiple tokens are collapsed together via normalization, they are indexed as one term, under the normalized form. However, we later relax this definition when discussing classification and clustering in Chapters 13–18, where there is no index. In these chapters, we drop the require ment of inclusion in the dictionary. A *term* means a normalized word.

Online edition (c)2009 Cambridge UP

*24 2 The term vocabulary and postings lists*

Boolean or free text queries, you always want to do the exact same tokeniza

tion of document and query words, generally by processing queries with the

same tokenizer. This guarantees that a sequence of characters in a text will

always match the same sequence typed in a query.3

These issues of tokenization are language-specific. It thus requires the lan

LANGUAGE guage of the document to be known. *Language identification* based on clas IDENTIFICATION sifiers that use short character subsequences as features is highly effective; most languages have distinctive signature patterns (see page 46 for refer

ences).

For most languages and particular domains within them there are unusual

specific tokens that we wish to recognize as terms, such as the programming

languages C++ and C#, aircraft names like B-52, or a T.V. show name such

as M\*A\*S\*H – which is sufficiently integrated into popular culture that you

find usages such as *M\*A\*S\*H-style hospitals*. Computer technology has in

troduced new types of character sequences that a tokenizer should probably

tokenize as a single token, including email addresses (jblack@mail.yahoo.com),

web URLs (http://stuff.big.com/new/specials.html),numeric IP addresses (142.32.48.231), package tracking numbers (1Z9999W99845399981), and more. One possible

solution is to omit from indexing tokens such as monetary amounts, num

bers, and URLs, since their presence greatly expands the size of the vocab

ulary. However, this comes at a large cost in restricting what people can

search for. For instance, people might want to search in a bug database for

the line number where an error occurs. Items such as the date of an email,

which have a clear semantic type, are often indexed separately as document

metadata (see Section 6.1, page 110).

HYPHENS In English, *hyphenation* is used for various purposes ranging from split ting up vowels in words (*co-education*) to joining nouns as names (*Hewlett*

*Packard*) to a copyediting device to show word grouping (*the hold-him-back*

*and-drag-him-away maneuver*). It is easy to feel that the first example should be

regarded as one token (and is indeed more commonly written as just *coedu*

*cation*), the last should be separated into words, and that the middle case is

unclear. Handling hyphens automatically can thus be complex: it can either

be done as a classification problem, or more commonly by some heuristic

rules, such as allowing short hyphenated prefixes on words, but not longer

hyphenated forms.

Conceptually, splitting on white space can also split what should be re

garded as a single token. This occurs most commonly with names (*San Fran*

*cisco, Los Angeles*) but also with borrowed foreign phrases (*au fait*) and com

3. For the free text case, this is straightforward. The Boolean case is more complex: this tok

enization may produce multiple terms from one query word. This can be handled by combining

the terms with an AND or as a phrase query (see Section 2.4, page 39). It is harder for a system

to handle the opposite case where the user entered as two terms something that was tokenized

together in the document processing.

Online edition (c)2009 Cambridge UP

*2.2 Determining the vocabulary of terms 25*

pounds that are sometimes written as a single word and sometimes space separated (such as *white space* vs. *whitespace*). Other cases with internal spaces that we might wish to regard as a single token include phone numbers ((800) 234- 2333) and dates (Mar 11, 1983). Splitting tokens on spaces can cause bad retrieval results, for example, if a search for York University mainly returns documents containing *New York University*. The problems of hyphens and non-separating whitespace can even interact. Advertisements for air fares frequently contain items like *San Francisco-Los Angeles*, where simply doing whitespace splitting would give unfortunate results. In such cases, issues of tokenization interact with handling phrase queries (which we discuss in Sec tion 2.4 (page 39)), particularly if we would like queries for all of *lowercase*, *lower-case* and *lower case* to return the same results. The last two can be han dled by splitting on hyphens and using a phrase index. Getting the first case right would depend on knowing that it is sometimes written as two words and also indexing it in this way. One effective strategy in practice, which is used by some Boolean retrieval systems such as Westlaw and Lexis-Nexis (Example 1.1), is to encourage users to enter hyphens wherever they may be possible, and whenever there is a hyphenated form, the system will general ize the query to cover all three of the one word, hyphenated, and two word forms, so that a query for over-eager will search for over-eager OR “over eager” OR overeager. However, this strategy depends on user training, since if you query using either of the other two forms, you get no generalization.

Each new language presents some new issues. For instance, French has a

variant use of the apostrophe for a reduced definite article ‘the’ before a word beginning with a vowel (e.g., *l’ensemble*) and has some uses of the hyphen with postposed clitic pronouns in imperatives and questions (e.g., *donne moi* ‘give me’). Getting the first case correct will affect the correct indexing of a fair percentage of nouns and adjectives: you would want documents mentioning both *l’ensemble* and *un ensemble* to be indexed under *ensemble*. Other languages make the problem harder in new ways. German writes

COMPOUNDS *compound nouns* without spaces (e.g., *Computerlinguistik* ‘computational lin guistics’; *Lebensversicherungsgesellschaftsangestellter* ‘life insurance company employee’). Retrieval systems for German greatly benefit from the use of a

COMPOUND-SPLITTER *compound-splitter* module, which is usually implemented by seeing if a word can be subdivided into multiple words that appear in a vocabulary. This phe nomenon reaches its limit case with major East Asian Languages (e.g., Chi nese, Japanese, Korean, and Thai), where text is written without any spaces between words. An example is shown in Figure 2.3. One approach here is to

WORD SEGMENTATION perform *word segmentation* as prior linguistic processing. Methods of word segmentation vary from having a large vocabulary and taking the longest vocabulary match with some heuristics for unknown words to the use of machine learning sequence models, such as hidden Markov models or condi tional random fields, trained over hand-segmented words (see the references

Online edition (c)2009 Cambridge UP

*26 2 The term vocabulary and postings lists*

! " # $ % & '

' ( ) \*  + , # - . /

◮ **Figure 2.3** The standard unsegmented form of Chinese text using the simplified characters of mainland China. There is no whitespace between words, not even be tween sentences – the apparent space after the Chinese period (◦) is just a typograph ical illusion caused by placing the character on the left side of its square box. The first sentence is just words in Chinese characters with no spaces between them. The second and third sentences include Arabic numerals and punctuation breaking up the Chinese characters.

◮ **Figure 2.4** Ambiguities in Chinese word segmentation. The two characters can be treated as one word meaning ‘monk’ or as a sequence of two words meaning ‘and’ and ‘still’.

a an and are as at be by for from has he in is it its of on that the to was were will with

◮ **Figure 2.5** A stop list of 25 semantically non-selective words which are common in Reuters-RCV1.

in Section 2.5). Since there are multiple possible segmentations of character sequences (see Figure 2.4), all such methods make mistakes sometimes, and so you are never guaranteed a consistent unique tokenization. The other ap proach is to abandon word-based indexing and to do all indexing via just short subsequences of characters (character *k*-grams), regardless of whether particular sequences cross word boundaries or not. Three reasons why this approach is appealing are that an individual Chinese character is more like a syllable than a letter and usually has some semantic content, that most words are short (the commonest length is 2 characters), and that, given the lack of standardization of word breaking in the writing system, it is not always clear where word boundaries should be placed anyway. Even in English, some cases of where to put word boundaries are just orthographic conventions – think of *notwithstanding* vs. *not to mention* or *into* vs. *on to* – but people are educated to write the words with consistent use of spaces.

Online edition (c)2009 Cambridge UP

*2.2 Determining the vocabulary of terms 27* **2.2.2 Dropping common terms: stop words**

Sometimes, some extremely common words which would appear to be of little value in helping select documents matching a user need are excluded STOP WORDS from the vocabulary entirely. These words are called *stop words*. The general COLLECTION strategy for determining a stop list is to sort the terms by *collection frequency* FREQUENCY (the total number of times each term appears in the document collection), and then to take the most frequent terms, often hand-filtered for their se mantic content relative to the domain of the documents being indexed, as STOP LIST a *stop list*, the members of which are then discarded during indexing. An example of a stop list is shown in Figure 2.5. Using a stop list significantly reduces the number of postings that a system has to store; we will present some statistics on this in Chapter 5 (see Table 5.1, page 87). And a lot of the time not indexing stop words does little harm: keyword searches with terms like the and by don’t seem very useful. However, this is not true for phrase searches. The phrase query “President of the United States”, which con tains two stop words, is more precise than President AND “United States”. The meaning of flights to London is likely to be lost if the word to is stopped out. A search for Vannevar Bush’s article *As we may think* will be difficult if the first three words are stopped out, and the system searches simply for documents containing the word think. Some special query types are disproportionately affected. Some song titles and well known pieces of verse consist entirely of words that are commonly on stop lists (*To be or not to be*, *Let It Be*, *I don’t want to be*, . . . ).

The general trend in IR systems over time has been from standard use of quite large stop lists (200–300 terms) to very small stop lists (7–12 terms) to no stop list whatsoever. Web search engines generally do not use stop lists. Some of the design of modern IR systems has focused precisely on how we can exploit the statistics of language so as to be able to cope with common words in better ways. We will show in Section 5.3 (page 95) how good compression techniques greatly reduce the cost of storing the postings for common words. Section 6.2.1 (page 117) then discusses how standard term weighting leads to very common words having little impact on doc ument rankings. Finally, Section 7.1.5 (page 140) shows how an IR system with impact-sorted indexes can terminate scanning a postings list early when weights get small, and hence common words do not cause a large additional processing cost for the average query, even though postings lists for stop words are very long. So for most modern IR systems, the additional cost of including stop words is not that big – neither in terms of index size nor in terms of query processing time.

Online edition (c)2009 Cambridge UP

*28 2 The term vocabulary and postings lists*

**Query term Terms in documents that should be matched**

Windows Windows

windows Windows, windows, window

window window, windows

◮ **Figure 2.6** An example of how asymmetric expansion of query terms can usefully model users’ expectations.

**2.2.3 Normalization (equivalence classing of terms)**

Having broken up our documents (and also our query) into tokens, the easy case is if tokens in the query just match tokens in the token list of the doc ument. However, there are many cases when two character sequences are not quite the same but you would like a match to occur. For instance, if you search for *USA*, you might hope to also match documents containing *U.S.A*.

TOKEN *Token normalization* is the process of canonicalizing tokens so that matches NORMALIZATION occur despite superficial differences in the character sequences of the to kens.4

EQUIVALENCE CLASSES The most standard way to normalize is to implicitly create *equivalence classes*, which are normally named after one member of the set. For instance, if the tokens *anti-discriminatory* and *antidiscriminatory* are both mapped onto the term antidiscriminatory, in both the document text and queries, then searches

for one term will retrieve documents that contain either.

The advantage of just using mapping rules that remove characters like hy

phens is that the equivalence classing to be done is implicit, rather than being fully calculated in advance: the terms that happen to become identical as the result of these rules are the equivalence classes. It is only easy to write rules of this sort that remove characters. Since the equivalence classes are implicit, it is not obvious when you might want to add characters. For instance, it would be hard to know to turn *antidiscriminatory* into *anti-discriminatory*.

An alternative to creating equivalence classes is to maintain relations be

tween unnormalized tokens. This method can be extended to hand-constructed lists of synonyms such as *car* and *automobile*, a topic we discuss further in Chapter 9. These term relationships can be achieved in two ways. The usual way is to index unnormalized tokens and to maintain a query expansion list of multiple vocabulary entries to consider for a certain query term. A query term is then effectively a disjunction of several postings lists. The alterna tive is to perform the expansion during index construction. When the doc ument contains automobile, we index it under car as well (and, usually, also vice-versa). Use of either of these methods is considerably less efficient than equivalence classing, as there are more postings to store and merge. The first

4. It is also often referred to as *term normalization*, but we prefer to reserve the name *term* for the output of the normalization process.

Online edition (c)2009 Cambridge UP

*2.2 Determining the vocabulary of terms 29*

method adds a query expansion dictionary and requires more processing at query time, while the second method requires more space for storing post ings. Traditionally, expanding the space required for the postings lists was seen as more disadvantageous, but with modern storage costs, the increased flexibility that comes from distinct postings lists is appealing.

These approaches are more flexible than equivalence classes because the expansion lists can overlap while not being identical. This means there can be an asymmetry in expansion. An example of how such an asymmetry can be exploited is shown in Figure 2.6: if the user enters windows, we wish to allow matches with the capitalized *Windows* operating system, but this is not plausible if the user enters window, even though it is plausible for this query to also match lowercase *windows*.

The best amount of equivalence classing or query expansion to do is a fairly open question. Doing some definitely seems a good idea. But doing a lot can easily have unexpected consequences of broadening queries in unin tended ways. For instance, equivalence-classing *U.S.A.* and *USA* to the latter by deleting periods from tokens might at first seem very reasonable, given the prevalent pattern of optional use of periods in acronyms. However, if I put in as my query term *C.A.T.*, I might be rather upset if it matches every appearance of the word *cat* in documents.5

Below we present some of the forms of normalization that are commonly employed and how they are implemented. In many cases they seem helpful, but they can also do harm. In fact, you can worry about many details of equivalence classing, but it often turns out that providing processing is done consistently to the query and to documents, the fine details may not have much aggregate effect on performance.

**Accents and diacritics.** Diacritics on characters in English have a fairly marginal status, and we might well want *cliché* and *cliche* to match, or *naive* and *naïve*. This can be done by normalizing tokens to remove diacritics. In many other languages, diacritics are a regular part of the writing system and distinguish different sounds. Occasionally words are distinguished only by their accents. For instance, in Spanish, *peña* is ‘a cliff’, while *pena* is ‘sorrow’. Nevertheless, the important question is usually not prescriptive or linguistic but is a question of how users are likely to write queries for these words. In many cases, users will enter queries for words without diacritics, whether for reasons of speed, laziness, limited software, or habits born of the days when it was hard to use non-ASCII text on many computer systems. In these cases, it might be best to equate all words to a form without diacritics.

5. At the time we wrote this chapter (Aug. 2005), this was actually the case on Google: the top result for the query *C.A.T.* was a site about cats, the Cat Fanciers Web Site http://www.fanciers.com/.

Online edition (c)2009 Cambridge UP

*30 2 The term vocabulary and postings lists*

CASE-FOLDING **Capitalization/case-folding.** A common strategy is to do *case-folding* by re ducing all letters to lower case. Often this is a good idea: it will allow in stances of *Automobile* at the beginning of a sentence to match with a query of *automobile*. It will also help on a web search engine when most of your users type in *ferrari* when they are interested in a *Ferrari* car. On the other hand, such case folding can equate words that might better be kept apart. Many proper nouns are derived from common nouns and so are distinguished only by case, including companies (*General Motors*, *The Associated Press*), govern ment organizations (*the Fed* vs. *fed*) and person names (*Bush*, *Black*). We al ready mentioned an example of unintended query expansion with acronyms, which involved not only acronym normalization (*C.A.T.* → *CAT*) but also case-folding (*CAT* → *cat*).

For English, an alternative to making every token lowercase is to just make

some tokens lowercase. The simplest heuristic is to convert to lowercase words at the beginning of a sentence and all words occurring in a title that is all uppercase or in which most or all words are capitalized. These words are usually ordinary words that have been capitalized. Mid-sentence capitalized words are left as capitalized (which is usually correct). This will mostly avoid case-folding in cases where distinctions should be kept apart. The same task can be done more accurately by a machine learning sequence model which uses more features to make the decision of when to case-fold. This is known

TRUECASING as *truecasing*. However, trying to get capitalization right in this way probably doesn’t help if your users usually use lowercase regardless of the correct case of words. Thus, lowercasing everything often remains the most practical solution.

**Other issues in English.** Other possible normalizations are quite idiosyn cratic and particular to English. For instance, you might wish to equate *ne’er* and *never* or the British spelling *colour* and the American spelling *color*. Dates, times and similar items come in multiple formats, presenting addi tional challenges. You might wish to collapse together *3/12/91* and *Mar. 12, 1991*. However, correct processing here is complicated by the fact that in the U.S., *3/12/91* is *Mar. 12, 1991*, whereas in Europe it is *3 Dec 1991*.

**Other languages.** English has maintained a dominant position on the WWW; approximately 60% of web pages are in English (Gerrand 2007). But that still leaves 40% of the web, and the non-English portion might be expected to grow over time, since less than one third of Internet users and less than 10% of the world’s population primarily speak English. And there are signs of change: Sifry (2007) reports that only about one third of blog posts are in English.

Other languages again present distinctive issues in equivalence classing.

Online edition (c)2009 Cambridge UP

*2.2 Determining the vocabulary of terms 31*

! " ! # $ % & ' ( ) \* + , - . / 0 )

1 2 3 4 5 6 7 & + 8 9 : ; : < = > ? @ A B C - D E

6 8 9 : ; : < ) F G \* H I \* : J) K + L M N ? O P Q R

S T U V V W X Y & Z [ N ? ) + \ ] ; ^ \_ + ` 4 a + b

; c d e \* f V g h V - ? i N j k l m n : A o p N 5 +

q V r s t u & v w x )Q y z {h | & } ~ M ? @ A

◮ **Figure 2.7** Japanese makes use of multiple intermingled writing systems and, like Chinese, does not segment words. The text is mainly Chinese characters with the hiragana syllabary for inflectional endings and function words. The part in latin letters is actually a Japanese expression, but has been taken up as the name of an environmental campaign by 2004 Nobel Peace Prize winner Wangari Maathai. His name is written using the katakana syllabary in the middle of the first line. The first four characters of the final line express a monetary amount that we would want to match with ¥500,000 (500,000 Japanese yen).

The French word for *the* has distinctive forms based not only on the gender (masculine or feminine) and number of the following noun, but also depend ing on whether the following word begins with a vowel: *le*, *la*, *l’*, *les*. We may well wish to equivalence class these various forms of *the*. German has a con vention whereby vowels with an umlaut can be rendered instead as a two vowel digraph. We would want to treat *Schütze* and *Schuetze* as equivalent.

Japanese is a well-known difficult writing system, as illustrated in Fig ure 2.7. Modern Japanese is standardly an intermingling of multiple alpha bets, principally Chinese characters, two syllabaries (hiragana and katakana) and western characters (Latin letters, Arabic numerals, and various sym bols). While there are strong conventions and standardization through the education system over the choice of writing system, in many cases the same word can be written with multiple writing systems. For example, a word may be written in katakana for emphasis (somewhat like italics). Or a word may sometimes be written in hiragana and sometimes in Chinese charac ters. Successful retrieval thus requires complex equivalence classing across the writing systems. In particular, an end user might commonly present a query entirely in hiragana, because it is easier to type, just as Western end users commonly use all lowercase.

Document collections being indexed can include documents from many different languages. Or a single document can easily contain text from mul tiple languages. For instance, a French email might quote clauses from a contract document written in English. Most commonly, the language is de tected and language-particular tokenization and normalization rules are ap plied at a predetermined granularity, such as whole documents or individual paragraphs, but this still will not correctly deal with cases where language changes occur for brief quotations. When document collections contain mul-

Online edition (c)2009 Cambridge UP

*32 2 The term vocabulary and postings lists*

tiple languages, a single index may have to contain terms of several lan guages. One option is to run a language identification classifier on docu ments and then to tag terms in the vocabulary for their language. Or this tagging can simply be omitted, since it is relatively rare for the exact same character sequence to be a word in different languages.

When dealing with foreign or complex words, particularly foreign names,

the spelling may be unclear or there may be variant transliteration standards giving different spellings (for example, *Chebyshev* and *Tchebycheff* or *Beijing* and *Peking*). One way of dealing with this is to use heuristics to equiva lence class or expand terms with phonetic equivalents. The traditional and best known such algorithm is the Soundex algorithm, which we cover in Section 3.4 (page 63).

**2.2.4 Stemming and lemmatization**

For grammatical reasons, documents are going to use different forms of a word, such as *organize*, *organizes*, and *organizing*. Additionally, there are fami lies of derivationally related words with similar meanings, such as *democracy*, *democratic*, and *democratization*. In many situations, it seems as if it would be useful for a search for one of these words to return documents that contain another word in the set.

The goal of both stemming and lemmatization is to reduce inflectional

forms and sometimes derivationally related forms of a word to a common base form. For instance:

am, are, is ⇒ be

car, cars, car’s, cars’ ⇒ car

The result of this mapping of text will be something like:

the boy’s cars are different colors ⇒

the boy car be differ color

STEMMING However, the two words differ in their flavor. *Stemming* usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the re

LEMMATIZATION moval of derivational affixes. *Lemmatization* usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base

LEMMA or dictionary form of a word, which is known as the *lemma*. If confronted with the token *saw*, stemming might return just *s*, whereas lemmatization would attempt to return either *see* or *saw* depending on whether the use of the token was as a verb or a noun. The two may also differ in that stemming most commonly collapses derivationally related words, whereas lemmatiza tion commonly only collapses the different inflectional forms of a lemma.

Online edition (c)2009 Cambridge UP

*2.2 Determining the vocabulary of terms 33*

Linguistic processing for stemming or lemmatization is often done by an ad ditional plug-in component to the indexing process, and a number of such components exist, both commercial and open-source.

The most common algorithm for stemming English, and one that has re PORTER STEMMER peatedly been shown to be empirically very effective, is *Porter’s algorithm* (Porter 1980). The entire algorithm is too long and intricate to present here, but we will indicate its general nature. Porter’s algorithm consists of 5 phases of word reductions, applied sequentially. Within each phase there are var ious conventions to select rules, such as selecting the rule from each rule group that applies to the longest suffix. In the first phase, this convention is used with the following rule group:

(2.1) **Rule Example**

SSES → SS caresses → caress

IES → I ponies → poni

SS → SS caress → caress

S → cats → cat

Many of the later rules use a concept of the *measure* of a word, which loosely checks the number of syllables to see whether a word is long enough that it is reasonable to regard the matching portion of a rule as a suffix rather than as part of the stem of a word. For example, the rule:

(*m* > 1) EMENT →

would map *replacement* to *replac*, but not *cement* to *c*. The official site for the Porter Stemmer is:

http://www.tartarus.org/˜martin/PorterStemmer/

Other stemmers exist, including the older, one-pass Lovins stemmer (Lovins 1968), and newer entrants like the Paice/Husk stemmer (Paice 1990); see:

http://www.cs.waikato.ac.nz/˜eibe/stemmers/

http://www.comp.lancs.ac.uk/computing/research/stemming/

Figure 2.8 presents an informal comparison of the different behaviors of these stemmers. Stemmers use language-specific rules, but they require less know ledge than a lemmatizer, which needs a complete vocabulary and morpho logical analysis to correctly lemmatize words. Particular domains may also require special stemming rules. However, the exact stemmed form does not matter, only the equivalence classes it forms.

LEMMATIZER Rather than using a stemmer, you can use a *lemmatizer*, a tool from Nat ural Language Processing which does full morphological analysis to accu rately identify the lemma for each word. Doing full morphological analysis produces at most very modest benefits for retrieval. It is hard to say more,

Online edition (c)2009 Cambridge UP

*34 2 The term vocabulary and postings lists*

***Sample text:*** Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of

expression that is more biologically transparent and accessible to

interpretation

***Lovins stemmer:*** such an analys can reve featur that ar not eas vis from th

vari in th individu gen and can lead to a pictur of expres that is mor

biolog transpar and acces to interpres

***Porter stemmer:*** such an analysi can reveal featur that ar not easili visibl

from the variat in the individu gene and can lead to a pictur of express

that is more biolog transpar and access to interpret

***Paice stemmer:*** such an analys can rev feat that are not easy vis from the

vary in the individ gen and can lead to a pict of express that is mor

biolog transp and access to interpret

◮ **Figure 2.8** A comparison of three stemming algorithms on a sample text.

because either form of normalization tends not to improve English informa tion retrieval performance in aggregate – at least not by very much. While it helps a lot for some queries, it equally hurts performance a lot for others. Stemming increases recall while harming precision. As an example of what can go wrong, note that the Porter stemmer stems all of the following words:

*operate operating operates operation operative operatives operational*

to oper. However, since *operate* in its various forms is a common verb, we would expect to lose considerable precision on queries such as the following with Porter stemming:

operational AND research

operating AND system

operative AND dentistry

For a case like this, moving to using a lemmatizer would not completely fix the problem because particular inflectional forms are used in particular col locations: a sentence with the words *operate* and *system* is not a good match for the query operating AND system. Getting better value from term normaliza tion depends more on pragmatic issues of word use than on formal issues of linguistic morphology.

The situation is different for languages with much more morphology (such

as Spanish, German, and Finnish). Results in the European CLEF evaluations have repeatedly shown quite large gains from the use of stemmers (and com pound splitting for languages like German); see the references in Section 2.5.

Online edition (c)2009 Cambridge UP

*2.2 Determining the vocabulary of terms 35 ?***Exercise 2.1** [⋆] Are the following statements true or false?

a. In a Boolean retrieval system, stemming never lowers precision.

b. In a Boolean retrieval system, stemming never lowers recall.

c. Stemming increases the size of the vocabulary.

d. Stemming should be invoked at indexing time but not while processing a query.

**Exercise 2.2** [⋆]

Suggest what normalized form should be used for these words (including the word itself as a possibility):

a. ’Cos

b. Shi’ite

c. cont’d

d. Hawai’i

e. O’Rourke

**Exercise 2.3** [⋆]

The following pairs of words are stemmed to the same form by the Porter stemmer. Which pairs would you argue shouldn’t be conflated. Give your reasoning.

a. abandon/abandonment

b. absorbency/absorbent

c. marketing/markets

d. university/universe

e. volume/volumes

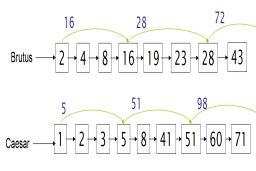
**Exercise 2.4** [⋆] For the Porter stemmer rule group shown in (2.1):

a. What is the purpose of including an identity rule such as SS → SS? b. Applying just this rule group, what will the following words be stemmed to? *circus canaries boss*

c. What rule should be added to correctly stem *pony*?

d. The stemming for *ponies* and *pony* might seem strange. Does it have a deleterious effect on retrieval? Why or why not?

Online edition (c)2009 Cambridge UP

*36 2 The term vocabulary and postings lists* ◮ **Figure 2.9 **Postings lists with skip pointers. The postings intersection can use a skip pointer when the end point is still less than the item on the other list.

**2.3 Faster postings list intersection via skip pointers**

In the remainder of this chapter, we will discuss extensions to postings list data structures and ways to increase the efficiency of using postings lists. Re call the basic postings list intersection operation from Section 1.3 (page 10): we walk through the two postings lists simultaneously, in time linear in the total number of postings entries. If the list lengths are *m* and *n*, the intersec tion takes *O*(*m* + *n*) operations. Can we do better than this? That is, empiri cally, can we usually process postings list intersection in sublinear time? We can, if the index isn’t changing too fast.

SKIP LIST One way to do this is to use a *skip list* by augmenting postings lists with skip pointers (at indexing time), as shown in Figure 2.9. Skip pointers are effectively shortcuts that allow us to avoid processing parts of the postings list that will not figure in the search results. The two questions are then where to place skip pointers and how to do efficient merging using skip pointers.

Consider first efficient merging, with Figure 2.9 as an example. Suppose

we’ve stepped through the lists in the figure until we have matched 8 on each list and moved it to the results list. We advance both pointers, giving us 16 on the upper list and 41 on the lower list. The smallest item is then the element 16 on the top list. Rather than simply advancing the upper pointer, we first check the skip list pointer and note that 28 is also less than 41. Hence we can follow the skip list pointer, and then we advance the upper pointer to 28 . We thus avoid stepping to 19 and 23 on the upper list. A number of variant versions of postings list intersection with skip pointers is possible depending on when exactly you check the skip pointer. One version is shown

Online edition (c)2009 Cambridge UP

*2.3 Faster postings list intersection via skip pointers 37*

INTERSECTWITHSKIPS(*p*1, *p*2)

1 *answer* ← h i

2 **while** *p*1 6= NIL and *p*2 6= NIL

3 **do if** *docID*(*p*1) = *docID*(*p*2)

4 **then** ADD(*answer*, *docID*(*p*1))

5 *p*1 ← *next*(*p*1)

6 *p*2 ← *next*(*p*2)

7 **else if** *docID*(*p*1) < *docID*(*p*2)

8 **then if** *hasSkip*(*p*1) and (*docID*(*skip*(*p*1)) ≤ *docID*(*p*2)) 9 **then while** *hasSkip*(*p*1) and (*docID*(*skip*(*p*1)) ≤ *docID*(*p*2)) 10 **do** *p*1 ← *skip*(*p*1)

11 **else** *p*1 ← *next*(*p*1)

12 **else if** *hasSkip*(*p*2) and (*docID*(*skip*(*p*2)) ≤ *docID*(*p*1)) 13 **then while** *hasSkip*(*p*2) and (*docID*(*skip*(*p*2)) ≤ *docID*(*p*1)) 14 **do** *p*2 ← *skip*(*p*2)

15 **else** *p*2 ← *next*(*p*2)

16 **return** *answer*

◮ **Figure 2.10** Postings lists intersection with skip pointers.

in Figure 2.10. Skip pointers will only be available for the original postings lists. For an intermediate result in a complex query, the call *hasSkip*(*p*) will always return false. Finally, note that the presence of skip pointers only helps for AND queries, not for OR queries.

Where do we place skips? There is a tradeoff. More skips means shorter skip spans, and that we are more likely to skip. But it also means lots of comparisons to skip pointers, and lots of space storing skip pointers. Fewer skips means few pointer comparisons, but then long skip spans which means that there will be fewer opportunities to skip. A simple heuristic for placing skips, which has been found to work well in practice, is that for a postings

list of length *P*, use √*P* evenly-spaced skip pointers. This heuristic can be improved upon; it ignores any details of the distribution of query terms. Building effective skip pointers is easy if an index is relatively static; it is harder if a postings list keeps changing because of updates. A malicious deletion strategy can render skip lists ineffective.

Choosing the optimal encoding for an inverted index is an ever-changing game for the system builder, because it is strongly dependent on underly ing computer technologies and their relative speeds and sizes. Traditionally, CPUs were slow, and so highly compressed techniques were not optimal. Now CPUs are fast and disk is slow, so reducing disk postings list size dom inates. However, if you’re running a search engine with everything in mem-

Online edition (c)2009 Cambridge UP

*38 2 The term vocabulary and postings lists*

ory then the equation changes again. We discuss the impact of hardware parameters on index construction time in Section 4.1 (page 68) and the im pact of index size on system speed in Chapter 5.

*?***Exercise 2.5** [⋆] Why are skip pointers not useful for queries of the form *x* OR *y*?

**Exercise 2.6** [⋆]

We have a two-word query. For one term the postings list consists of the following 16 entries:

[4,6,10,12,14,16,18,20,22,32,47,81,120,122,157,180]

and for the other it is the one entry postings list:

[47].

Work out how many comparisons would be done to intersect the two postings lists with the following two strategies. Briefly justify your answers:

a. Using standard postings lists

b. Using postings lists stored with skip pointers, with a skip length of √*P*, as sug gested in Section 2.3.

**Exercise 2.7** [⋆] Consider a postings intersection between this postings list, with skip pointers:

3 5 9 15 24 39 60 68 75 81 84 89 92 96 97 100 115

and the following intermediate result postings list (which hence has no skip pointers): 3 5 89 95 97 99 100 101

Trace through the postings intersection algorithm in Figure 2.10 (page 37).

a. How often is a skip pointer followed (i.e., *p*1is advanced to *skip*(*p*1))?

b. How many postings comparisons will be made by this algorithm while intersect ing the two lists?

c. How many postings comparisons would be made if the postings lists are inter sected without the use of skip pointers?

Online edition (c)2009 Cambridge UP

*2.4 Positional postings and phrase queries 39*

**2.4 Positional postings and phrase queries**

Many complex or technical concepts and many organization and product names are multiword compounds or phrases. We would like to be able to pose a query such as Stanford University by treating it as a phrase so that a sentence in a document like *The inventor Stanford Ovshinsky never went to uni versity.* is not a match. Most recent search engines support a double quotes

PHRASE QUERIES syntax (“stanford university”) for *phrase queries*, which has proven to be very easily understood and successfully used by users. As many as 10% of web queries are phrase queries, and many more are implicit phrase queries (such as person names), entered without use of double quotes. To be able to sup port such queries, it is no longer sufficient for postings lists to be simply lists of documents that contain individual terms. In this section we consider two approaches to supporting phrase queries and their combination. A search engine should not only support phrase queries, but implement them effi ciently. A related but distinct concept is term proximity weighting, where a document is preferred to the extent that the query terms appear close to each other in the text. This technique is covered in Section 7.2.2 (page 144) in the context of ranked retrieval.

**2.4.1 Biword indexes**

One approach to handling phrases is to consider every pair of consecutive terms in a document as a phrase. For example, the text *Friends, Romans,* BIWORD INDEX *Countrymen* would generate the *biwords*:

friends romans

romans countrymen

In this model, we treat each of these biwords as a vocabulary term. Being able to process two-word phrase queries is immediate. Longer phrases can be processed by breaking them down. The query stanford university palo alto can be broken into the Boolean query on biwords:

“stanford university” AND “university palo” AND “palo alto”

This query could be expected to work fairly well in practice, but there can and will be occasional false positives. Without examining the documents, we cannot verify that the documents matching the above Boolean query do actually contain the original 4 word phrase.

Among possible queries, nouns and noun phrases have a special status in describing the concepts people are interested in searching for. But related nouns can often be divided from each other by various function words, in phrases such as *the abolition of slavery* or *renegotiation of the constitution*. These needs can be incorporated into the biword indexing model in the following

Online edition (c)2009 Cambridge UP

*40 2 The term vocabulary and postings lists*

way. First, we tokenize the text and perform part-of-speech-tagging.6 We can then group terms into nouns, including proper nouns, (N) and function words, including articles and prepositions, (X), among other classes. Now deem any string of terms of the form NX\*N to be an extended biword. Each such extended biword is made a term in the vocabulary. For example:

renegotiation of the constitution

N X X N

To process a query using such an extended biword index, we need to also parse it into N’s and X’s, and then segment the query into extended biwords, which can be looked up in the index.

This algorithm does not always work in an intuitively optimal manner

when parsing longer queries into Boolean queries. Using the above algo rithm, the query

cost overruns on a power plant

is parsed into

“cost overruns” AND “overruns power” AND “power plant”

whereas it might seem a better query to omit the middle biword. Better results can be obtained by using more precise part-of-speech patterns that define which extended biwords should be indexed.

The concept of a biword index can be extended to longer sequences of

words, and if the index includes variable length word sequences, it is gen PHRASE INDEX erally referred to as a *phrase index*. Indeed, searches for a single term are not naturally handled in a biword index (you would need to scan the dic tionary for all biwords containing the term), and so we also need to have an index of single-word terms. While there is always a chance of false positive matches, the chance of a false positive match on indexed phrases of length 3 or more becomes very small indeed. But on the other hand, storing longer phrases has the potential to greatly expand the vocabulary size. Maintain ing exhaustive phrase indexes for phrases of length greater than two is a daunting prospect, and even use of an exhaustive biword dictionary greatly expands the size of the vocabulary. However, towards the end of this sec tion we discuss the utility of the strategy of using a partial phrase index in a compound indexing scheme.

6. Part of speech taggers classify words as nouns, verbs, etc. – or, in practice, often as finer grained classes like “plural proper noun”. Many fairly accurate (c. 96% per-tag accuracy) part of-speech taggers now exist, usually trained by machine learning methods on hand-tagged text. See, for instance, Manning and Schütze (1999, ch. 10).

Online edition (c)2009 Cambridge UP

*2.4 Positional postings and phrase queries 41*

to, 993427:

h 1, 6: h7, 18, 33, 72, 86, 231i;

2, 5: h1, 17, 74, 222, 255i;

4, 5: h8, 16, 190, 429, 433i;

5, 2: h363, 367i;

7, 3: h13, 23, 191i; . . .i

be, 178239:

h 1, 2: h17, 25i;

4, 5: h17, 191, 291, 430, 434i;

5, 3: h14, 19, 101i; . . .i

◮ **Figure 2.11** Positional index example. The word to has a document frequency 993,477, and occurs 6 times in document 1 at positions 7, 18, 33, etc.

**2.4.2 Positional indexes**

For the reasons given, a biword index is not the standard solution. Rather, POSITIONAL INDEX a *positional index* is most commonly employed. Here, for each term in the vocabulary, we store postings of the form docID: hposition1, position2, . . .i, as shown in Figure 2.11, where each position is a token index in the docu ment. Each posting will also usually record the term frequency, for reasons discussed in Chapter 6.

To process a phrase query, you still need to access the inverted index en tries for each distinct term. As before, you would start with the least frequent term and then work to further restrict the list of possible candidates. In the merge operation, the same general technique is used as before, but rather than simply checking that both terms are in a document, you also need to check that their positions of appearance in the document are compatible with the phrase query being evaluated. This requires working out offsets between the words.

✎ **Example 2.1: Satisfying phrase queries.** Suppose the postings lists for to and

be are as in Figure 2.11, and the query is “to be or not to be”. The postings lists to access are: to, be, or, not. We will examine intersecting the postings lists for to and be. We first look for documents that contain both terms. Then, we look for places in the lists where there is an occurrence of *be* with a token index one higher than a position of *to*, and then we look for another occurrence of each word with token index 4 higher than the first occurrence. In the above lists, the pattern of occurrences that is a possible match is:

to: h. . . ; 4:h. . . ,429,433i; . . .i

be: h. . . ; 4:h. . . ,430,434i; . . .i

Online edition (c)2009 Cambridge UP

*42 2 The term vocabulary and postings lists*

POSITIONALINTERSECT(*p*1, *p*2, *k*)

1 *answer* ← h i

2 **while** *p*1 6= NIL and *p*2 6= NIL

3 **do if** *docID*(*p*1) = *docID*(*p*2)

4 **then** *l* ← h i

5 *pp*1 ← *positions*(*p*1)

6 *pp*2 ← *positions*(*p*2)

7 **while** *pp*1 6= NIL

8 **do while** *pp*2 6= NIL

9 **do if** |*pos*(*pp*1) − *pos*(*pp*2)| ≤ *k*

10 **then** ADD(*l*, *pos*(*pp*2))

11 **else if** *pos*(*pp*2) > *pos*(*pp*1)

12 **then break**

13 *pp*2 ← *next*(*pp*2)

14 **while** *l* 6= h i and |*l*[0] − *pos*(*pp*1)| > *k*

15 **do** DELETE(*l*[0])

16 **for each** *ps* ∈ *l*

17 **do** ADD(*answer*,h*docID*(*p*1), *pos*(*pp*1), *ps*i)

18 *pp*1 ← *next*(*pp*1)

19 *p*1 ← *next*(*p*1)

20 *p*2 ← *next*(*p*2)

21 **else if** *docID*(*p*1) < *docID*(*p*2)

22 **then** *p*1 ← *next*(*p*1)

23 **else** *p*2 ← *next*(*p*2)

24 **return** *answer*

◮ **Figure 2.12** An algorithm for proximity intersection of postings lists *p*1 and *p*2. The algorithm finds places where the two terms appear within *k* words of each other and returns a list of triples giving docID and the term position in *p*1 and *p*2.

The same general method is applied for within *k* word proximity searches,

of the sort we saw in Example 1.1 (page 15):

employment /3 place

Here, /*k* means “within *k* words of (on either side)”. Clearly, positional in dexes can be used for such queries; biword indexes cannot. We show in Figure 2.12 an algorithm for satisfying within *k* word proximity searches; it is further discussed in Exercise 2.12.

**Positional index size.** Adopting a positional index expands required post ings storage significantly, even if we compress position values/offsets as we

Online edition (c)2009 Cambridge UP

*2.4 Positional postings and phrase queries 43*

will discuss in Section 5.3 (page 95). Indeed, moving to a positional index also changes the asymptotic complexity of a postings intersection operation, because the number of items to check is now bounded not by the number of documents but by the total number of tokens in the document collection *T*. That is, the complexity of a Boolean query is Θ(*T*) rather than Θ(*N*). How ever, most applications have little choice but to accept this, since most users now expect to have the functionality of phrase and proximity searches.

Let’s examine the space implications of having a positional index. A post ing now needs an entry for each occurrence of a term. The index size thus depends on the average document size. The average web page has less than 1000 terms, but documents like SEC stock filings, books, and even some epic poems easily reach 100,000 terms. Consider a term with frequency 1 in 1000 terms on average. The result is that large documents cause an increase of two orders of magnitude in the space required to store the postings list:

Expected Expected entries

Document size postings in positional posting

1000 1 1

100,000 1 100

While the exact numbers depend on the type of documents and the language being indexed, some rough rules of thumb are to expect a positional index to be 2 to 4 times as large as a non-positional index, and to expect a compressed positional index to be about one third to one half the size of the raw text (after removal of markup, etc.) of the original uncompressed documents. Specific numbers for an example collection are given in Table 5.1 (page 87) and Table 5.6 (page 103).

**2.4.3 Combination schemes**

The strategies of biword indexes and positional indexes can be fruitfully combined. If users commonly query on particular phrases, such as Michael Jackson, it is quite inefficient to keep merging positional postings lists. A combination strategy uses a phrase index, or just a biword index, for certain queries and uses a positional index for other phrase queries. Good queries to include in the phrase index are ones known to be common based on re cent querying behavior. But this is not the only criterion: the most expensive phrase queries to evaluate are ones where the individual words are com mon but the desired phrase is comparatively rare. Adding *Britney Spears* as a phrase index entry may only give a speedup factor to that query of about 3, since most documents that mention either word are valid results, whereas adding *The Who* as a phrase index entry may speed up that query by a factor of 1000. Hence, having the latter is more desirable, even if it is a relatively less common query.

Online edition (c)2009 Cambridge UP