Name of the Paper: Natural Language Processing
Paper Code: CS702B
Contact (Periods/Week):=3L/Week
Credit Point: 3
No. of Lectures: 41

Prerequisite:

1. A solid background in mathematics, including probability, set theory.
2. Determination to learn new and difficult things.

Course Objective(s)

- To learn the basics of NLP.
- To learn the principles and application of different NLP techniques.
- To learn the details of NLP algorithms, different tools and knowing their use.

Course Outcome(s)

On completion of the course students will be able to

CS702B.1 Uunderstand the fundamental concept of NLP, Regular Expression, Finite State Automata along with the concept and application of word tokenization, normalization, sentence segmentation, word extraction, spell checking in the context of NLP.

CS702B.2 Understand the concept of Morphology such as Inflectional and Derivational Morphology and different morphological parsing techniques including FSTs.

CS702B.3 Understand the concepts related to language modeling with introduction to N-grams, chain rule, smoothing, Witten Bell discounting, backoff, deleted interpolation, spelling and word prediction and their evaluation along with the concept of Markov chain, HMM, Forward and Viterbi algorithm, POS tagging.

CS702B.4 Understand the concept of different text classification techniques, sentiment analysis, concepts related to CFG in the context of NLP.

CS702B.5 Understand the concept of lexical semantics, lexical dictionary such as WordNet, lexical computational semantics, distributional word similarity and concepts related to the field of Information Retrieval in the context of NLP.

Module I: [12L]

Introduction to NLP [2L]

Human languages, models, definition of NLP, text representation in computers, encoding schemes, issues and strategies, application domain, tools for NLP, Linguistic organisation of NLP, phase in natural language processing, applications
.

Regular Expression and Automata [2L]
Finite State Automata. Introduction to CFG and different parsing technniques.

Tokenization [4L]
Word Tokenization, Normalization, Sentence Segmentation, Named Entity Recognition,Multi Word Extraction, Spell Checking – Bayesian Approach, Minimum Edit Distance

Morphology [4L]
Morphology – Inflectional and Derivational Morphology, Finite State Morphological Parsing, The Lexicon and Morphotactics, Morphological Parsing with Finite State Transducers, Orthographic Rules and Finite State Transducers, Porter Stemmer.

Module II: [9L]

Language Modeling [4L]
Introduction to N-grams, Chain Rule, Smoothing – Add-One Smoothing, Witten-Bell Discounting; Backoff, Deleted. Interpolation, N-grams for Spelling and Word Prediction, Evaluation of language models. Spelling errors, detection and elimination using probabilistic models, pronunciation variation (lexical, allophonic, dialect), decision tree model, counting words in Corpora

Hidden Markov Models and POS Tagging [5L]
Markov Chain, Hidden Markov Models, Forward Algorithm, Viterbi Algorithm, Part of Speech Tagging – Rule based and Machine Learning based approaches, concept of HMM tagger Evaluation. Handling of unknown words, named entities, multi word expressions.

Module III: [9L]
Text Classification [4L]
Text Classification, Naïve Bayes' Text Classification, Evaluation, Sentiment Analysis – Opinion Mining and Emotion Analysis, Resources and Techniques.

Context Free Grammar [5L]
Context Free Grammar and Constituency, Some common CFG phenomena for English, Top-Down and Bottom-up parsing, Probabilistic Context Free Grammar, Dependency Parsing

Module IV: [11L]

Computational Lexical Semantics [5L]
Introduction to Lexical Semantics – Homonymy, Polysemy, Synonymy, Thesaurus – WordNet, VerbNet, Computational Lexical Semantics – Thesaurus based and Distributional Word Similarity, Lexemes (homonymy, polysemy, synonymy, hyponymy), word structure, metaphor, metonymy. Word sense disambiguation, machine learning approaches, dictionary based approaches.

Information Retrieval [6L]
Boolean Retrieval, Term-document incidence, The Inverted Index, Query Optimization, Phrase Queries, Ranked Retrieval , Term Frequency and Inverse Document Frequency based ranking, Zone Indexing, Query term proximity, Cosine ranking, Combining different features for ranking, Search Engine Evaluation, Relevance Feedback.
Resource management with XML, Management of linguistic data with the help of GATE, NLTK

Text books:
1. D. Jurafsky & J. H. Martin – "Speech and Language Processing – An introduction to Language processing, Computational Linguistics, and Speech Recognition",Pearson Education
2. Chris Manning and Hinrich Schütze, "Foundations of Statistical Natural Language Processing", MIT Press. Cambridge, MA: May 1999.

Reference books:

1. Allen, James. 1995. – “Natural Language Understanding”. Benjamin/Cummings, 2ed.
2. Bharathi, A., Vineet Chaitanya and Rajeev Sangal. 1995. Natural Language Processing- “A Pananian Perspective”. Prentice Hll India, Eastern Economy Edition.
3. Siddiqui T., Tiwary U. S.. “Natural language processing and Information retrieval”, OUP, 2008.
4. Eugene Cherniak: “Statistical Language Learning”, MIT Press, 1993.
5. Manning, Christopher and Heinrich Schütze. 1999. “Foundations of Statistical Natural Language Processing”. MIT Press.

CO-PO Mapping

CO	PO1	PO2	POP3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	CO	PO1	PO2
CS702B.1		2	3	3	3						3	2	3	2	3
CS702B.2		2	2	2	1						3	2	3	2	2
CS702B.3		2	3	3	3	3					3	2	3	2	2
CS702B.4		2	2	2	1						3	2	3	2	2
CS702B.5	3				2	3	1						2	2	3
CS702B	3	2	3	3	2	3	1	-	-	-	3	2	3	2	2

LESSON PLAN

Module No.	Lecture No.	Topics	Reference Books
1 Introduction to NLP	L1	Human languages, models, definition of NLP, text representation in computers, encoding schemes	T1,T2, R1,R2
	L2	Issues and strategies, application domain, tools for NLP, Linguistic organisation of NLP, phase in natural language processing, applications	
	L3	Finite State Automata	
	L4	Introduction to CFG and different parsing technniques	
	L5	Word Tokenization, Normalization	
	L6	Sentence Segmentation, Named Entity Recognition	
	L7	Multi Word Extraction	
	L8	Spell Checking — Bayesian Approach, Minimum Edit Distance	
	L9	Morphology — Inflectional and Derivational Morphology	
	L10	Finite State Morphological Parsing	
	L11	The Lexicon and Morphotactics, Morphological Parsing with Finite State Transducers	
	L12	Orthographic Rules and Finite State Transducers, Porter Stemmer	
2 **Language Modelling**	L1	Introduction to N-grams, Chain Rule	T1,T2, R1,R2
	L2	Smoothing — Add-One Smoothing, Witten-Bell Discounting	
	L3	Backoff, Deleted. Interpolation, N-grams for Spelling and Word Prediction, Evaluation of language models. Spelling errors	
	L4	Detection and elimination using probabilistic models, pronunciation variation (lexical, allophonic, dialect), decision tree model, counting words in Corpora	
	L5	Markov Chain, Hidden Markov Models	
	L6	Forward Algorithm, Viterbi Algorithm, Part of Speech Tagging	
	L7	Rule based and Machine Learning based approaches. concept of HMM tagger Evaluation	
	L8	Handling of unknown words	
	L9	Named entities, multi word expressions	
3 Text Classification	L1	Text Classification, Naive Bayes' Text Classification	T1,T2, R3,R2
	L2	Evaluation	
	L3	Sentiment Analysis — Opinion Mining - and	

Module No.	Lecture No.	Topics	Reference Books
		Emotion Analysis	
	L4	Resources and Techniques	
	L5	Context Free Grammar and Constituency	
	L6	Some common CFG phenomena for English	
	L7	Top-Down and Bottom-up parsing	
	L8	Probabilistic Context Free Grammar	
	L9	Dependency Parsing	
4 Computational Lexical Semantics	L1	Introduction to Lexical Semantics	T1,T2, R4,R5
	L2	Homonymy, Polysemy, Synonymy, Thesaurus, WordNet, VerbNet	
	L3	Computational Lexical Semantics — Thesaurus based and Distributional Word Similarity	
	L4	word structure, metaphor, metonymy	
	L5	Word sense disambiguation, machine learning approaches, dictionary based approaches	
	L6	Boolean Retrieval, Term-document incidence	
	L7	The Inverted Index. Query Optimization	
	L8	Phrase Queries. Ranked Retrieval	
	L9	Term Frequency and Inverse Document Frequency based ranking	
	L10	Zone Indexing. Query term proximity, Cosine ranking, Combining different features for ranking	
	L11	Search Engine Evaluation, Relevance Feedback. Resource management with XML, Management of linguistic data with the help of GATE, NLTK	

LECTURE No.:1

INTRODUCTION to NLP

Natural language processing (NLP) is a collective term referring to automaticcomputational processing of human languages. This includes both algorithms thattake human-produced text as input, and algorithms that produce natural lookingtext as outputs. One of the earliest goals for computers was the automatic translation of text from one languageto another. Automatic or machine translation is perhaps one of the most challenging artificialintelligence tasks given the fluidity of human language. Classically, rule-based systems wereused for this task, which were replaced in the 1990s with statistical methods. More recently,deep neural network models achieve state-of-the-art results in a field that is aptly named neuralmachine translation.

DEFINITION

Natural Language Processing, or NLP for short, is broadly de_ned as the automatic manipulation of natural language, like speech and text, by software. The study of natural language processing has been around for more than 50 years and grew out of the _eld of linguistics with the rise of computers.

Natural language refers to the way we, humans, communicate with each other. Namely, speech and text. We are surrounded by text.
. Signs
. Menus
. Email
. SMS
. Web Pages
. and so much more...

Natural Language Processing (NLP) refers to AI method of communicating with an intelligent systems using a natural language such as English.

Processing of Natural Language is required when you want an intelligent system like robot to perform as per your instructions, when you want to hear decision from a dialogue based clinical expert system, etc.

The field of NLP involves making computers to perform useful tasks with the natural languages humans use. The input and output of an NLP system can be −

- Speech
- Written Text

Representing text

When any key on a keyboard is pressed, it needs to be converted into a binary number so that it can be processed by the computer and the typed character can appear on the screen.

A code where each number represents a character can be used to convert **text** into binary.

A code where each number represents a character can be used to convert text into binary. One code we can use for this is called ASCII. The ASCII code takes each character on the keyboard and assigns it a binary number. For example:

- the letter ‘a’ has the binary number 0110 0001 (this is the denary number 97)
- the letter ‘b’ has the binary number 0110 0010 (this is the denary number 98)
- the letter ‘c’ has the binary number 0110 0011 (this is the denary number 99)

Text characters start at denary number 0 in the ASCII code, but this covers special characters including punctuation, the return key and control characters as well as the number keys, capital letters and lower case letters.

ASCII code can only store 128 characters, which is enough for most words in English but not enough for other languages. If you want to use accents in European languages or larger alphabets such as Cyrillic (the Russian alphabet) and Chinese Mandarin then more characters are needed. Therefore another code, called Unicode, was created. This meant that computers could be used by people using different languages.

Encoding Scheme:

Different encoding schemes can represent character data. We can incorporate EBCDIC, ASCII, and Unicode encoding schemes in Db2® ODBC.

The EBCDIC and ASCII encoding scheme include multiple code pages; each code page represents 256 characters for one specific geography or one generic geography. The Unicode encoding scheme does not require the use of code pages, because it represents over 65,000 characters. Unicode can also accommodate many different languages and geographies.

The Unicode standard defines several implementations including UTF-8, UCS-2, UTF-16, and UCS-4. ODBC Db2 supports Unicode in the following formats:

- UTF-8 (variable length, 1-byte to 6-byte characters)
- UCS-2 (2-byte characters)

Unicode and ASCII are alternatives to the EBCDIC character encoding scheme. The Db2® ODBC driver supports input and output character string arguments to ODBC APIs and input and output host variable data in each of these encoding schemes.

With this support, we can manipulate data, SQL statements, and API string arguments in EBCDIC, Unicode, or ASCII.

- Types of encoding schemes
Different encoding schemes can represent character data. We can incorporate EBCDIC,

ASCII, and Unicode encoding schemes in Db2 ODBC.

- Application programming guidelines for handling different encoding schemes
  The Db2 ODBC driver determines whether an application is an EBCDIC, Unicode, or ASCII application by evaluating the CURRENTAPPENSCH keyword in the initialization file. We must compile our application with a compiler option that corresponds to this setting.
- Suffix-W API function syntax
  Db2 for z/OS® supports function prototypes for suffix-W APIs, which have slight differences from generic APIs.

- Examples of handling the application encoding scheme
  We can use the application encoding scheme to bind a UCS-2 result set column, bind UTF-8 data to parameter markers, retrieve UTF-8 data into application

variables. The application encoding scheme also allows you to use suffix-W APIs. To perform these tasks, you must declare variables, specifying data types appropriately for a particular encoding scheme.

Application domain

An **application domain** is a mechanism (similar to a process in an operating system) used within the Common Language Infrastructure (CLI) to isolate executed software applications from one another so that they do not affect each other. Each application domain has its own virtual address space which scopes the resources for the application domain using that address space.

Lecture No. : 2

Tools for NLP:

Nowadays, a lot of tools have been published to do natural language processing jobs. The following are tools which provide functions such as parsing, finding topic automatically, etc.

- OpenNLP: a Java package to do text tokenization, part-of-speech tagging, chunking, etc. (tutorial)
- Stanford Parser: a Java implementation of probabilistic natural language parsers, both highly optimized PCFG* and lexicalized dependency parsers, and a lexicalized PCFG parser
- ScalaNLP: Natural Language Processing and machine learning.
- Snowball: a stemmer, support C and Java.
- MALLET: a Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications to text.
- JGibbLDA: LDA in Java
- Apache Lucene Core: a Java library for stop-words removal and stemming
- Stanford Topic Modelling Toolbox: CVB0 algorithm, etc.

Linguistic organisation of NLP:

The **Association for Neuro-Linguistic Programming (ANLP)** is a UK organisation founded in 1985 by Frank Kevlin to promote neuro-linguistic programming (NLP). Since 2005, the organisation is led by Karen Moxom as Managing Director.

Associate membership is open to anyone interested in NLP and Professional membership is open to holders of recognised NLP qualifications, which include a minimum number of face to face training hours.

The ANLP publishes *Rapport* magazine quarterly as both a PDF and in print format, which is available to non-members by subscription, and has published the proceedings of a NLP Research Conference held at the University of Surrey in 2008. A second conference was held at Cardiff University in 2010 and a third at the University of Hertfordshire in 2012. Volumes 2 and 3 of the Research Journal have subsequently been published (in 2011 and 2013 respectively).

ANLP were awarded Small Business of the Year in the Hertfordshire Business Awards 2009.

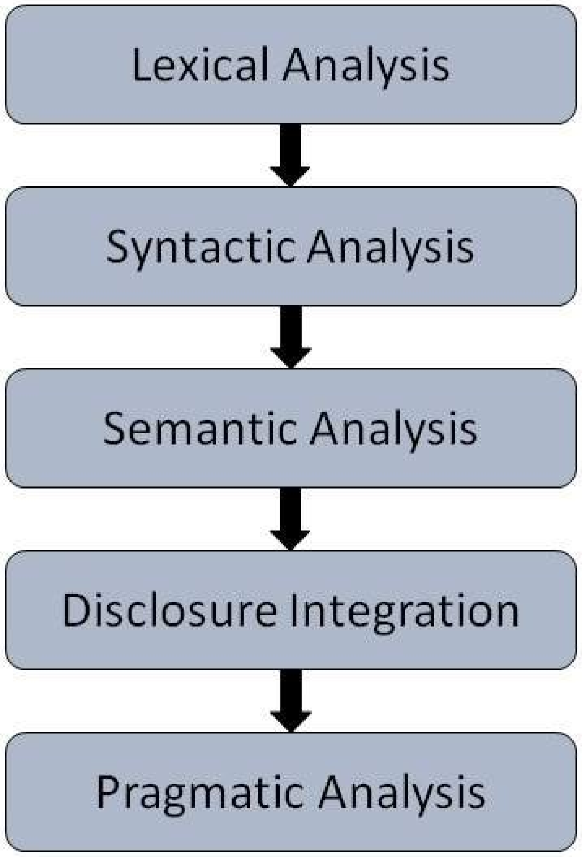
It was a member organisation of the United Kingdom Council for Psychotherapy (UKCP) to 2002 when the role was taken over by its daughter organization, the Neuro Linguistic Psychotherapy and Counselling Association (NLPtCA). The NLPtCA is a founder member of the European Association for Neuro-Linguistic Psychotherapy, a European wide accrediting organisation (EWAO) for NLPt within the European Association for Psychotherapy (EAP).

Phase in NLP:

Steps in NLP

There are general five steps −

- **Lexical Analysis** − It involves identifying and analyzing the structure of words. Lexicon of a language means the collection of words and phrases in a language. Lexical analysis is dividing the whole chunk of txt into paragraphs, sentences, and words.
- **Syntactic Analysis (Parsing)** − It involves analysis of words in the sentence for grammar and arranging words in a manner that shows the relationship among the words. The sentence such as “The school goes to boy” is rejected by English syntactic analyzer.



- **Semantic Analysis** − It draws the exact meaning or the dictionary meaning from the text. The text is checked for meaningfulness. It is done by mapping syntactic structures and objects in the task domain. The semantic analyzer disregards sentence such as "hot ice-cream".
- **Discourse Integration** − The meaning of any sentence depends upon the meaning of the sentence just before it. In addition, it also brings about the meaning of immediately succeeding sentence.
- **Pragmatic Analysis** − During this, what was said is re-interpreted on what it actually meant. It involves deriving those aspects of language which require real world knowledge.

Applications:

NLP is the process of producing meaningful phrases and sentences in the form of natural language from some internal representation.

It involves −

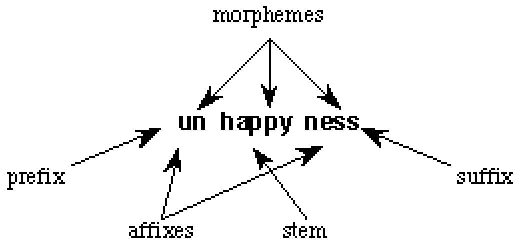
- **Text planning** − It includes retrieving the relevant content from knowledge base.

- **Sentence planning** − It includes choosing required words, forming meaningful phrases, setting tone of the sentence.
- **Text Realization** − It is mapping sentence plan into sentence structure.

LECTURE No.:3

Morphology:

Morphology is the study of the structure and formation of words. Its most important unit is the *morpheme*, which is defined as the "minimal unit of meaning". (Linguistics textbooks usually define it slightly differently as "the minimal unit of grammatical analysis".) Consider a word like: "unhappiness". This has three parts:



There are three morphemes, each carrying a certain amount of meaning. *un* means "not", while *ness* means "being in a state or condition". *Happy* is a *free morpheme* because it can appear on its own (as a "word" in its own right). *Bound morphemes* have to be attached to a free morpheme, and so cannot be words in their own right. Thus we can't have sentences in English such as "Jason feels very un ness today".

The important thing about this example is to notice how the morphemes all represent a "unit of meaning" and how they remain absolutely identifiable within the structure of the words. This is in contrast to what happens in the last class: the inflecting languages.

Inflecting languages:

The words in inflecting languages do show different forms and it is possible to break the words into smaller units and label them, in the same way that the Turkish example was presented above. However, the result is a very muddled and contradictory account. Usual examples are based on Latin and rely on a knowledge of the Latin grammatical case example, which most English undergraduates don't have. As a simple example, the Latin for "I love" is *amo.* This is means that the ending *o* is used to express the meanings, *first person* ("I" or "we"), *singular, present tense,* and also other meanings.

This classification has only three classes. Is it really possible to fit all the world's languages into three classes? From one way of looking at the problem, it is impossible to fit any of the languages into any of the classes, because each language is impure.

That is to say, if you look hard enough, you will find inflection in mainly agglutinative languages, inflection in isolating languages, agglutination in inflectional languages and so on.

Morphological forms of regular verbs:

stem	Walk	merge	try	map
-s form	Walks	merges	tries	maps
-ing principle	Walking	merging	trying	mapping
Past form or –ed participle	Walked	merged	tried	mapped

These regular verbs and forms are significant in the morphology of English because of their majority and being productive.

Survey of (Mostly) English Morphology Inflectional Morphology–Morphological forms of irregular verbs

stem	eat	catch	cut
-s form	eats	catches	cuts
-ing principle	eating	catching	cutting
Past form	Ate	caught	cut
–ed participlee	Aten	caught	cut

Survey of (Mostly) English Morphology Derivational Morphology Nominalization in English: –The formation of new nouns, often from verbs or adjectives

Suffix	Base Verb/Adjective	Derived Noun
-action	computerize (V)	computerization
-ee	appoint (V)	appointee
-er	kill (V)	killer

-ness	fuzzy (A)	fuzziness

Morphological processes

In the example given above of *unhappiness,* we saw two kinds of affix, a prefix and a suffix. Just to show that languages do really vary greatly, there are also infixes. For instance the Bontoc language from the Philippines use an infix *um* to change adjectives and nouns into verbs. So the word *fikas,* which means "strong" is transformed into the verb "be strong" by the addition of the infix: *f-um-ikas.*

There are a number of morphological processes of which some are more important than others for NLP. The account given here is selective and unusual in that it points out the practical aspects of the processes selected.

Inflection

Inflection is the process of changing the form of a word so that it expresses information such as number, person, case, gender, tense, mood and aspect, but the syntactic category of the word remains unchanged. As an example, the plural form of the noun in English is usually formed from the singular form by adding an *s.*

- car / cars
- table / tables
- dog / dogs

In each of these cases, the syntactic category of the word remains unchanged.

It doesn't take long to find examples where the simple rule given above doesn't fit. So there are smaller groups of nouns that form plurals in different ways:

- wolf / wolves
- knife / knives
- switch / switches.

A little more thought and we can think of apparently completely irregular plural forms, such as:

- foot / feet
- child / children.

English verbs are relatively simple (especially compared with languages like Finnish which has over 12,000 verb inflections).

- mow - stem
- mow**s** - third person singular, present tense
- mow**ed** - past tense and past participle
- mow**ing** - present continuous tense

NLP aspects

Languages like French or German have much more inflection than English and so it is customary to include morphological analysers in systems that process these languages. NLP systems for English often don't include any morphological process, especially if they are small-scale systems. Where English-based systems do include analysis of inflection, the regular forms of words are analysed using one of the standard techniques (for instance, Finite State Automata), while the exceptions (the irregular words) are each listed individually. This means that regular forms have to be entered in the dictionary only once, which can save a lot of space and data entry if the dictionary holds a lot of syntactic and semantic information.

Derivation

Inflection doesn't change the syntactic category of a word. Derivation *does* change the category. Linguists classify derivation in English according to whether or not it induces a change of pronunciation. For instance, adding the suffix *ity* changes the pronunciation of the root of *active* so the stress is on the second syllable: *activity.* The addition of the suffix *al* to *approve* doesn't change the pronunciation of the root: *approval.*

NLP aspects

The obvious use of derivational morphology in NLP systems is to reduce the number of forms of words to be stored. So, if there is already an entry for the base form of the verb *sing,* then it should be possible to add rules to map the nouns *singer* and *singers* onto the same entry. The problem her is that the detection of the derivation of *singer* from *sing* must allow also allow the morphological analyser to contribute the information that is special to *singer.* This seems a little obscure, but an example will make it clearer. The addition of *er* to the word indicates that it is a person who is undertaking the action. This semantic information must be added to the stored information from the dictionary entry for the root form *sing* so that the correct meaning of the sentence can be found. This seems fine, but suppose the next two words are: *recorder* and *dragster*. The use of the *er* morpheme can't be taken to necessarily mean someone who undertakes the action represented by the root form.

Any linguistic processing is likely to have an effect of uncovering potential ambiguity, particularly where humans wouldn't expect it. Derivational morphological analysers can do this quite easily, because they are always attempting to reduce words to smaller units. So the word *really* might be analysed as both *really* (ie the word itself) and the as *re + ally*. This introduced ambiguity may be eliminated by later, syntactic processing but nonetheless, it does mean that there has to be more processing (and so a slower system).

Derivational morphology is particularly useful for Machine Translation(MT). Successful MT has to process large quantities of text which can contain many previously unseen words. Some words are neologisms (that is, newly-coined words). If the analyser can reduce these words to their base form, it may be able to translate that and, in effect, coin a new word in the target language by simply following rules. To give a couple of examples: neologisms often have a proper name as their root. A knowledge of how *Thatcherite* and *Majorism* were formed from proper names could enable an MT system to translate them into an idiomatic equivalent in the target language.

input: form
output: lemma + morphological features

Input	Output
cat	cat + N + sg
cats	cat + N + pl
cities	city + N + pl
merging	merge + V + pres_part
Caught	(catch + V + past) or (catch + V + past_part)

<u>LECTURE No.:4</u>

Finite State Morphological Parsing:

Morphological parsing, in NLP, is the process of determining the morphemes from which a given word is constructed. It must be able to distinguish between orthographic rules and morphological rules. For example, the word '**foxes**' can be decomposed into 'fox' (the stem), and 'es' (a suffix indicating plurality) i.e. breaking down words into components and building a structured representation.
English: **cats ->cat +N +Pl**
caught -> catch +V +Past

Morphology is the study of the way words are built up from smaller meaning-bearing units, morphemes. Two broad classes of morphemes: –The stems: the “main” morpheme of the word, supplying the main meaning, while –The affixes add “additional” meaning of various kinds.Affixes are further divided into prefixes, suffixes, infixes, and circumfixes.
Suffix: eat-s
Prefix: un-buckle
Circumfix: ge-sag-t (said) sagen (to say) (in German)
Infix: hingi (borrow) humingi (the agent of an action))in Philippine language Tagalog)

The generally accepted approach to morphological parsing is through the use of a finite state transducer (FST), which inputs words and outputs their stem and modifiers. The FST is initially created through algorithmic parsing of some word source, such as a dictionary, complete with modifier markups.

Orthographic rules are general rules used when breaking a word into its stem and modifiers. An example would be: singular English words ending with -y, when pluralized, end with -ies. Contrast this to Morphological rules which contain corner cases to these general rules. Both of these types of rules are used to construct systems that can do morphological parsing.
Morphological rules are exceptions to the orthographic rules used when breaking a word into its stem and modifiers. An example would be while one normally

pluralizes a word in English by adding 's' as a suffix, the word 'fish' does not change when pluralized. Contrast this to orthographic rules which contain general rules. Both of these types of rules are used to construct systems that can do morphological parsing. Applications of morphological processing are machine translation, spell checker, information retrieval.

LECTURE No.:5

Finite-state transducer

A **finite-state transducer** (**FST**) is a finite-state machine with two memory *tapes*, following the terminology for Turing machines: an input tape and an output tape. This contrasts with an ordinary finite-state automaton, which has a single tape. An FST is a type of finite-state automaton that maps between two sets of symbols. An FST is more general than a finite-state automaton (FSA). An FSA defines a formal language by defining a set of accepted strings while an FST defines relations between sets of strings.

An FST will read a set of strings on the input tape and generates a set of relations on the output tape. An FST can be thought of as a translator or relater between strings in a set.

In morphological parsing, an example would be inputting a string of letters into the FST, the FST would then output a string of morphemes.

Formally, a finite transducer T is a 6-tuple (Q, Σ, Γ, I, F, δ) such that:

- Q is a finite set, the set of *states*;
- Σ is a finite set, called the *input alphabet*;
- Γ is a finite set, called the *output alphabet*;
- I is a subset of Q, the set of *initial states*;
- F is a subset of Q, the set of *final states*; and
- δ (where ε is the empty string) is the *transition relation*.

We can view (Q, δ) as a labeled directed graph, known as the *transition graph* of T: the set of vertices is Q, and δ means that there is a labeled edge going from vertex q to vertex r. We also say that a is the *input label* and b the *output label* of that edge.

Operations on finite-state transducers

The following operations defined on finite automata also apply to finite transducers:

- Union. Given transducers *T* and *S*, there exists a transducer T U S such that x[T U S]y if and only if x[T]y or x[S]y .
- Concatenation. Given transducers *T* and *S*, there exists a transducer T . S such that

  x[T . S]y if and only if there exist x1,x2,y1,y2 with x=x1x2 and y=y1y2, x1[T]y1, x2[S]y2

- Kleene closure. Given a transducer *T*, there exists a transducer T* with the following properties:

  $^{\varepsilon}[T^*]^{\varepsilon}$
  w[T*]y and x[T]z then wx[T*]yz

  and x[T*]y does not hold unless mandated by (**k1**) or (**k2**).
- Composition. Given a transducer *T* on alphabets Σ and Γ and a transducer *S* on alphabets Γ and Δ, there exists a transducer T . S on Σ and Δ such that x[T . S]z if and only if there exists a string y € Γ* such that x[T]y and y[S]z . This operation extends to the weighted case.

  This definition uses the same notation used in mathematics for relation composition. However, the conventional reading for relation composition is the other way around: given two relations *T* and *S*, (x,z) € T . S when there exist some *y* such that
  (x,y) € S and (y,z) € T
  Projection to an automaton. There are two projection functions: π1 preserves the input tape, and π2 preserves the output tape. The first projection, π1 is defined as follows:
  Given a transducer *T*, there exists a finite automaton π1T such that π1T accepts *x* if and only if there exists a string *y* for which x[T]y
  The second projection, π2 is defined similarly.
- Determinization. Given a transducer *T*, we want to build an equivalent transducer that has a unique initial state and such that no two transitions leaving any state share the same input label. The powerset construction can be extended to transducers, or even weighted transducers, but sometimes fails to halt; indeed, some non-deterministic transducers do not admit equivalent deterministic transducers. Characterizations of determinizable transducers have been proposed along with efficient algorithms to test them: they rely on the semiring used in the weighted case as well as a general property on the structure of the transducer (the twins property).
- Weight pushing for the weighted case.
- Minimization for the weighted case.
- Removal of epsilon-transitions.

LECTURE No.:6

Porter Stemmer:

The Porter stemming algorithm is a process for removing suffixes from words in English. Removing suffixes automatically is an operation which is especially useful in the field of information retrieval. In a typical IR environment a document is represented by a vector of words, or terms. Terms with a common stem will usually have similar meanings for example:

CONNECT

CONNECTED

CONNECTING

CONNECTION

CONNECTIONS

Frequently, the performance of an IR system will be improved if term groups such as this are conflated into a single term. This may be done by removal of the various suffixes – ED, -ING, -ION, -IONS to leave the single term CONNECT. The suffix stripping process will reduce the total number of terms in the IR system, and hence reduce the size and complexity of the data in the system, which is always advantageous.

Porter's stemmer advantage is its simplicity and speed. Porter algorithm was made in the assumption that we don't have a stem dictionary and that the purpose of the task is to improve IR performance. The program is given an explicit list of suffixes, and with each suffix, the criterion under which it may be removed from a word to leave a valid stem.

ALGORITHM:

A consonant in a word is a letter other than A,E,I,O or U and other than Y preceded by a consonant. So in TOY the consonants are T and Y and in SYZYGY they are S, Z and G. If a letter is not a consonant it is a vowel.

A consonant will be denoted by c, a vowel by v. A list ccc…… of length greater than 0 will be denoted by C and a list vvv…. of length greater than 0 will be denoted by V. Any word, or part of a word, therefore has one of the four forms:

CVCV … C

CVCV … V

VCVC … C

VCVC … V

These may all be represented by the single form:

[C]VCVC … [V]

Where the square brackets denote arbitrary presence of their contents.

Using (VC)m to denote VC repeated m times, this may again be written as:

[C](VC){m}[V]

M will be called the measure of any word or word part when represented in this form. The case m=0 covers the null word.

LECTURE No.:7

Language Modelling:

A statistical **language model** is a probability distribution over sequences of words. Given such a sequence, say of length *m*, it assigns a probability P(w1,w2,…….,wm) to the whole sequence.

The language model provides context to distinguish between words and phrases that sound similar. For example, in American English, the phrases "recognize speech" and "wreck a nice beach" sound similar, but mean different things.

Data sparsity is a major problem in building language models. Most possible word sequences are not observed in training. One solution is to make the assumption that the probability of a word only depends on the previous *n* words. This is known as an *n*-gram model or unigram model when *n* = 1. The unigram model is also known as the bag of words model.

Estimating the relative likelihood of different phrases is useful in many natural language processing applications, especially those that generate text as an output. Language modeling is used in speech recognition, machine translation, part-of-speech tagging, parsing, Optical Character Recognition, handwriting recognition, information retrieval and other applications.

In speech recognition, sounds are matched with word sequences. Ambiguities are easier to resolve when evidence from the language model is integrated with a pronunciation model and an acoustic model.

Language models are used in information retrieval in the query likelihood model. There a separate language model is associated with each document in a collection. Documents are ranked based on the probability of the query *Q* in the document's language model $P(Q|M_d)$. Commonly, the unigram language model is used for this purpose.

Unigram

A unigram model can be treated as the combination of several one-state finite automata. It splits the probabilities of different terms in a context, e.g. from

$P(t_1t_2t_3)= P(t_1)P(t_2|t_1)P(t_3|t_1t_2)$ to $P_{uni}(t_1t_2t_3)= P(t_1)P(t_2)P(t_3)$

In this model, the probability of each word only depends on that word's own probability in the document, so we only have one-state finite automata as units. The automaton itself has a probability distribution over the entire vocabulary of the model, summing to 1.

- Language models are useful for NLP applications such as
  - Next word prediction
  - Machine translation
  - Spelling correction
  - Authorship Identification
  - Natural language generation

Chain rule:

In calculus, the **chain rule** is a formula for computing the derivative of the composition of two or more functions. That is, if f and g are functions, then the chain rule expresses the derivative of their composition $f \circ g$ (the function which maps x to $f(g(x))$) in terms of the derivatives of f and g and the product of functions as follows: $(f \circ g)'=(f' \circ g).g'$

This may equivalently be expressed in terms of the variable. Let $F = f \circ g$, or equivalently, $F(x) = f(g(x))$ for all x. Then one can also write $F'(x) = f'(g(x))g'(x)$

If a variable z depends on the variable y, which itself depends on the variable x, so that y and z are therefore dependent variables, then z, via the intermediate variable of y, depends on x as well. The chain rule then states,

$dz/dx=dz/dy.dy/dx$

The two versions of the chain rule are related; if $z=f(y)$ and $y=g(x)$, then

$$dz/dx=dz/dy.dy/dx=f'(y)g'(x)=f'(g(x))g'(x)$$

LECTURE No.:8

Smoothing:

Smoothing is the task of adjusting the maximum likelihood estimate of probabilities to produce more accurate probabilities.The name comes from the fact that these techniques tend to make distributions more uniform, by adjusting low probabilities such as zero probabilities upward, and high probabilities downward. Smoothing not only prevents zero probabilities, attempts to improves the accuracy of the model as a whole.

If one n-gram is unseen in the sentence, probability of the whole sentence becomes zero. To avoid this, some probability mass has to be reserved for the unseen words. So for avoiding this, we use Smoothing techniques. This zero probability problem also occurs in text categorization using *Multinomial Naïve Bayes* . Probability of a test document given some class can be zero even if a single word in that document is unseen

Add-one Smoothing:

11

Bigram Model

	I	Want	to	eat	Chinese	food	lunch
I	9	1088	1	14	1	1	1
Want	4	1	787	1	7	9	7
To	4	1	11	861	4	1	13
Eat	1	1	3	1	20	3	53
Chinese	3	1	1	1	1	121	2
Food	20	1	18	1	1	1	1
lunch	5	1	1	1	1	2	1

12

Raw Bigram Counts

	I	Want	to	eat	Chinese	food	lunch
I	8	1087	0	13	0	0	0
Want	3	0	786	0	6	8	6
To	3	0	10	860	3	0	12
Eat	0	0	2	0	19	2	52
Chinese	2	0	0	0	0	120	1
Food	19	0	17	0	0	0	0
lunch	4	0	0	0	0	1	0

7

Probability Space

	I	Want	to	eat	Chinese	food	lunch
I	.0023	.32	0	.0038	0	0	0
Want	.0025	0	.65	0	.0049	.0066	.0049
To	.00092	0	.0031	.26	.00092	0	.0037
Eat	0	0	.0021	0	.020	.0021	.055
Chinese	.0094	0	0	0	0	.56	.0047
Food	.013	0	.011	0	0	0	0
lunch	.0087	0	0	0	0	.0022	0

8

Concept of "Discounting"

13

Laplace Correction - Adjusted Counts

	I	Want	to	eat	Chinese	food	lunch
I	6	740	.68	10	.68	.68	.68
Want	2	.42	331	.42	3	4	3
To	3	.69	8	594	3	.69	9
Eat	.37	.37	1	.37	7.4	1	20
Chinese	.36	.12	.12	.12	.12	15	.24
Food	10	.48	9	.48	.48	.48	.48
lunch	1.1	.22	.22	.22	.22	.44	.22

14

Laplace Correction - Adjusted Counts

	I	Want	to	eat	Chinese	food	lunch
I	6	740	.68	10	.68	.68	.68
Want	2	.42	331	.42	3	4	3
To	3	.69	8	594	3	.69	9
Eat	.37	.37	1	.37	7.4	1	20
Chinese	.36	.12	.12	.12	.12	15	.24
Food	10	.48	9	.48	.48	.48	.48
lunch	1.1	.22	.22	.22	.22	.44	.22

14

Witten-Bell Smoothing

The probability of seeing a zero-frequency N-gram can be modeled by the probability of seeing an N-gram for the first time.

where T is the types we have already seen, and N is the number of tokens

16

Witten Bell - for Bigram

17

Smoothed counts

$$c_i^* = \begin{cases} \frac{T}{Z}\frac{N}{N+T}, if\ c_i = 0 \\ c_i\frac{N}{N+T}, if\ c_i > 0 \end{cases}$$

18

Witten Bell - Example

W	T(W)
I	95
Want	76
To	130
Eat	124
Chinese	20
Food	82
Lunch	45

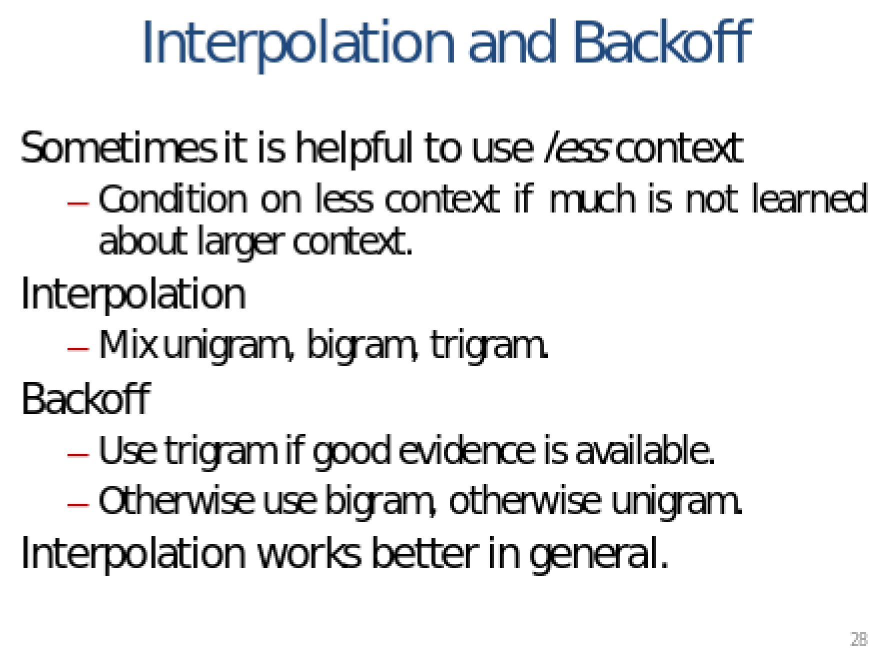
Z(w) = V - T(w)

19

Witten Bell – Smoothed Counts

	I	Want	to	eat	Chinese	food	lunch
I	8	1060	.062	13	.062	.062	.062
Want	3	.046	740	.046	6	8	6
To	3	.085	10	827	3	.085	12
Eat	.075	.075	2	.075	17	2	46
Chinese	2	.012	.012	.012	.012	109	1
Food	18	.059	16	.059	.059	.059	.059
lunch	4	.026	.026	.026	.026	1	.026

LECTURE No.:9



Interpolation and Backoff

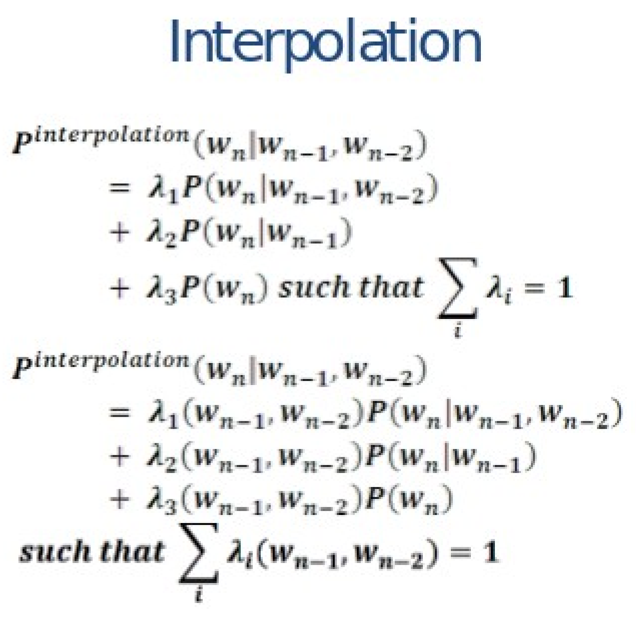
Sometimes it is helpful to use *less* context
- Condition on less context if much is not learned about larger context.

Interpolation
- Mix unigram, bigram, trigram.

Backoff
- Use trigram if good evidence is available.
- Otherwise use bigram, otherwise unigram.

Interpolation works better in general.

28

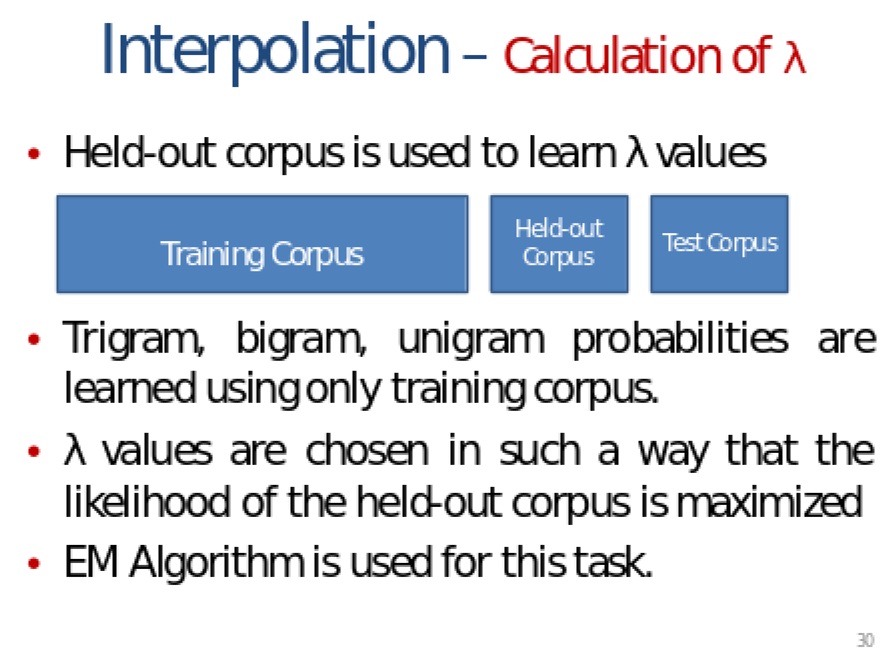


Interpolation

$$
\begin{aligned}
P^{interpolation}&(w_n|w_{n-1}, w_{n-2}) \\
&= \lambda_1 P(w_n|w_{n-1}, w_{n-2}) \\
&+ \lambda_2 P(w_n|w_{n-1}) \\
&+ \lambda_3 P(w_n) \ such\ that \sum_i \lambda_i = 1
\end{aligned}
$$

$$
\begin{aligned}
P^{interpolation}&(w_n|w_{n-1}, w_{n-2}) \\
&= \lambda_1(w_{n-1}, w_{n-2}) P(w_n|w_{n-1}, w_{n-2}) \\
&+ \lambda_2(w_{n-1}, w_{n-2}) P(w_n|w_{n-1}) \\
&+ \lambda_3(w_{n-1}, w_{n-2}) P(w_n)
\end{aligned}
$$

$$
such\ that \sum_i \lambda_i(w_{n-1}, w_{n-2}) = 1
$$



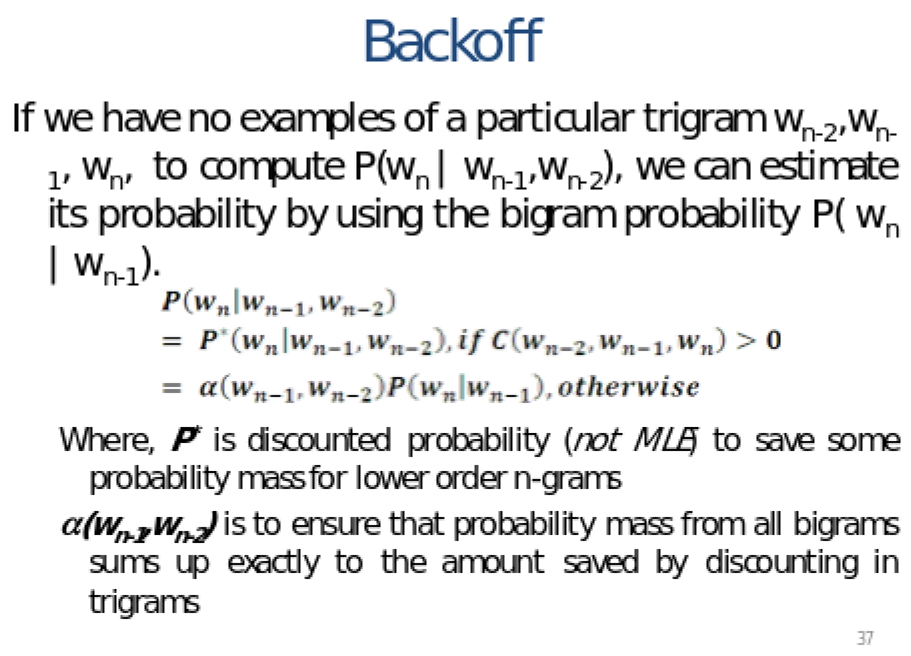
Interpolation – Calculation of λ

- Held-out corpus is used to learn λ values

Training Corpus	Held-out Corpus	Test Corpus

- Trigram, bigram, unigram probabilities are learned using only training corpus.
- λ values are chosen in such a way that the likelihood of the held-out corpus is maximized
- EM Algorithm is used for this task.

30



Backoff

If we have no examples of a particular trigram w_{n-2}, w_{n-1}, w_n, to compute $P(w_n \mid w_{n-1}, w_{n-2})$, we can estimate its probability by using the bigram probability $P(w_n \mid w_{n-1})$.

$$\begin{aligned} &P(w_n|w_{n-1},w_{n-2}) \\ &= P^*(w_n|w_{n-1},w_{n-2}), if\ C(w_{n-2},w_{n-1},w_n) > 0 \\ &= \alpha(w_{n-1},w_{n-2})P(w_n|w_{n-1}), otherwise \end{aligned}$$

Where, P^* is discounted probability (*not MLE*) to save some probability mass for lower order n-grams

$\alpha(w_{n-1},w_{n-2})$ is to ensure that probability mass from all bigrams sums up exactly to the amount saved by discounting in trigrams

37

Both interpolation (Jelinek-Mercer) and backoff (Katz) involve com-bining information from higher- and lower-order models.The difference is in determining the probability of n-grams with non-zerocounts, interpolated models use information from lower-order mod-els while backoff models do not.In both backoff and interpolated models, lower-order models areused in determining the probability ofn-grams with zero counts.It turns out that it’s not hard to create a backoff version of aninterpolated algorithm, and vice-versa.

Unigram ML model:

$p_{ML}(w_i) = c(w_i)/\sum_{w_i} c(w_i)$

Bigram interpolated model: $p_{interp}(w_i|w_{i-1}) = \lambda p_{ML}(w_i|w_{i-1}) + (1-\lambda) p_{ML}(w_i)$

LECTURE No.:10

n-gram:

In the fields of computational linguistics and probability, an ***n*-gram** is a contiguous sequence of *n* items from a given sample of text or speech. The items can be phonemes, syllables, letters, words or base pairs according to the application. The *n*-grams typically are collected from a text or speech corpus. When the items are words, *n*-grams may also be called *shingles*

Using Latin numerical prefixes, an *n*-gram of size 1 is referred to as a "unigram"; size 2 is a "bigram" (or, less commonly, a "digram"); size 3 is a "trigram". English cardinal numbers are sometimes used, e.g., "four-gram", "five-gram", and so on. In computational biology, a polymer or oligomer of a known size is called a *k*-mer instead of an *n*-gram, with specific names using Greek numerical prefixes such as "monomer", "dimer", "trimer", "tetramer", "pentamer", etc., or English cardinal numbers, "one-mer", "two-mer", "three-mer", etc.

An ***n*-gram model** is a type of probabilistic language model for predicting the next item in such a sequence in the form of a ($n - 1$)–order Markov model. *n*-gram models are now widely used in probability, communication theory, computational linguistics (for instance, statistical natural language processing), computational biology (for instance, biological sequence analysis), and data compression. Two benefits of *n*-gram models (and algorithms that use them) are simplicity and scalability – with larger *n*, a model can store more context with a well-understood space–time tradeoff, enabling small experiments to scale up efficiently.

n-gram models are widely used in statistical natural language processing. In speech recognition, phonemes and sequences of phonemes are modeled using a *n*-gram distribution. For parsing, words are modeled such that each *n*-gram is composed of *n* words. For language identification, sequences of characters/graphemes (*e.g.*, letters of the alphabet) are modeled for different languages.[4] For sequences of characters, the 3-grams (sometimes referred to as "trigrams") that can be generated from "good morning" are "goo", "ood", "od ", "d m", " mo", "mor" and so forth, counting the space character as a gram (sometimes the beginning and end of a text are modeled explicitly, adding "__g", "_go", "ng_", and "g__").

Evaluation of Language model:

A statistical **language model** is a probability distribution over sequences of words. Given such a sequence, say of length *m*, it assigns a probability to the whole sequence.

The language model provides context to distinguish between words and phrases that sound similar. For example, in American English, the phrases "recognize speech" and "wreck a nice beach" sound similar, but mean different things.

Data sparsity is a major problem in building language models. Most possible word sequences are not observed in training. One solution is to make the assumption that the probability of a word only depends on the previous *n* words. This is known as an *n*-gram model or unigram model when *n* = 1. The unigram model is also known as the bag of words model.

Estimating the relative likelihood of different phrases is useful in many natural language processing applications, especially those that generate text as an output. Language modeling is used in speech recognition, machine translation, part-of-speech tagging, parsing, Optical Character Recognition, handwriting recognition, information retrieval and other applications.

In speech recognition, sounds are matched with word sequences. Ambiguities are easier to resolve when evidence from the language model is integrated with a pronunciation model and an acoustic model.

Language models are used in information retrieval in the query likelihood model. There a separate language model is associated with each document in a collection. Documents are ranked based on the probability of the query *Q* in the document's language model $P(Q|M_d)$. Commonly, the unigram language model is used for this purpose.

Unigram

A unigram model can be treated as the combination of several one-state finite automata. It splits the probabilities of different terms in a context, e.g. from

$P(t_1t_2t_3) = P(t_1)P(t_2|t_1)P(t_3|t_1t_2)$ to $P_{uni}(t_1t_2t_3) = P(t_1)P(t_2)P(t_3)$.

In this model, the probability of each word only depends on that word's own probability in the document, so we only have one-state finite automata as units. The automaton itself has a probability distribution over the entire vocabulary of the model, summing to 1. The following is an illustration of a unigram model of a document.

Terms	Probability in doc
A	0.1
world	0.2
likes	0.05
We	0.05
share	0.3
...	...

$$\Sigma_{\text{term in doc}} P(\text{term}) = 1$$

The probability generated for a specific query is calculated as

$$P(\text{query}) = \pi_{\text{term in query}} \; P(\text{term})$$

Different documents have unigram models, with different hit probabilities of words in it. The probability distributions from different documents are used to generate hit probabilities for each query. Documents can be ranked for a query according to the probabilities. Example of unigram models of two documents:

Terms	Probability in Doc1	Probability in Doc2
A	0.1	0.3
world	0.2	0.1
likes	0.05	0.03
We	0.05	0.02
share	0.3	0.2
...	...	...

In information retrieval contexts, unigram language models are often smoothed to avoid instances where $P(\text{term}) = 0$. A common approach is to generate a maximum-likelihood model for the entire collection and linearly interpolate the collection model with a maximum-likelihood model for each document to smooth the model.

Decision tree model:

In computational complexity and communication complexity theories the **decision tree model** is the model of computation or communication in which an algorithm or communication process is considered to be basically a decision tree, i.e., a sequence of branching operations based on comparisons of some quantities, the comparisons being assigned the unit computational cost.

The branching operations are called "tests" or "queries". In this setting the algorithm in question may be viewed as a computation of a Boolean function f: $\{ 0,1 \}^n \rightarrow \{ 0,1 \}$ where the input is a series of queries and the output is the final decision. Every query is dependent on previous queries.

Several variants of decision tree models have been introduced, depending on the complexity of the operations allowed in the computation of a single comparison and the way of branching.

Decision trees models are instrumental in establishing lower bounds for computational complexity for certain classes of computational problems and algorithms: the lower bound for worst-case computational complexity is proportional to the largest depth among the decision trees for all possible inputs for a given computational problem. The computation complexity of a problem or an algorithm expressed in terms of the decision tree model is called **decision tree complexity** or **query complexity**.

Corpora:

In principle, any collection of more than one text can be called a corpus.

For e.g. **Shakespeare corpus.** Corpus in modern linguistic theories has more specific

connotations; it has four main characteristics:

Sampling and representativeness

Finite size (not always)

Machine-readable form

A standard reference

"Corpus" in language engineering tends to de-emphasize 1, 2 and 4.
The very first *modern* corpus: **Brown Corpus** (1967)

Word *token*: each word occurring in a text/corpus

Corpora sizes are measured as total number of words (=tokens)

Word *type*: unique words

Q: Are 'sleep' and 'sleeps' different types or the same type?

A: Depends.

Sometimes, types are meant as *lemma* types. Sometimes, inflected and derived words count as different type

<u>LECTURE No.:11</u>

Hidden markov model:

HMM Based POS Tagger:

The task of tagging is to find the sequence of tags T= {t_1,t_2,….,t_n} i.e. optimal for a word sequence W= {w_1,w_2,….,w_n}. Probabilistically, this can be framed as the problem of finding the most probable tag sequence for a word sequence i.e. argmaxT P(T|W). Apply Bayes' law i.e. P(T|W) = P(T) * P(W|T) / P(W). The probability of the word sequence P(W) will be the same regardless of the tag sequence chosen, i.e. to search for argmaxT P(T) * P(W|T). Probability of the tag P(T) is calculated as P(T) = P(t_1,t_2,….,t_n) = P(t_1) * P(t_2|t_1) * P(t_3|t_2) * ….* P(t_n|t_{n-1}) using Uni-gram model. Probability of P(W|T) is calculated as :

P(W|T) = P(w_1|t_1) * P(w_2|t_2) * P(w_3|t_3) * …* P(w_n|t_n) using Uni-gram model.

In the DataBase Table Three Columns are present.

1. Current Word
2. Current Tag
3. Previous Tag

Calculates the occurance of that word in the total train files.Calculate the current word tag values in the total train files. Here each word may have more than one tag, therefore in this case, selecting those tag which have highest frequency for that specified word.If more than one tag have the same frequency value, then assigning one tag for the specified word based on some hand written rules
Now calculate probability of each tag with it's previous tag and multiply them until reach the end part of the corpus, which are stored in the DataBase table with percentage from which we get P(T).
Calculate each word with it's own tag and multiply them from which we get P(W|T).Now multiply P(T) by P(W|T) and get the highest probability tag value.Finally, Tag the word in the test file(i.e. highest percentage value for the "word tag" which is learned and calculated by the train files).
Estimates three parameters directly from the tagged corpus.

$$P_{start}(t_i) = \frac{\text{no. of sentences which begin with } t_i}{\text{no. of sentences}}$$

$$P(t_i \mid t_{i-1}) = \frac{count(t_{i-1}t_i)}{count(t_{i-1})}$$

$$P(w_i \mid t_{i-1}) = \frac{count(w_i \text{ with } t_i)}{count(t_i)}$$

IIITH Tagset for Indian Languages:

All the tags used in this tagset are broadly classified into three Groups.

Group 1:
All tags in this group are similar to the Penn tagset. For e.g.
NN — Noun , NNP — Proper Nouns , PRP — Pronoun , VAUX — Verb Auxiliary , JJ — Adjective , RB — Adverb , RP — Particle , CC — Conjunction , UH — Interjection , SYM — Special Symbol.

Group 2:
This group includes those tags that are a modification of some tags in the Penn tagset. For e.g.
PREP — Postposition , QF — Quantifiers , QFNUM — Quantifiers Number , VFM — Verb Finite Main , VJJ — Verb Non-Finite Adjectival , VRB — Verb Non-finite Adverbial , VNN — Verb Non-Finite Nominal , QW — Question Words.

Group 3:
This designed to cater some phenomena that are specific to Indian languages. For e.g.
NLOC — Noun Location , INTF — Intensifier , NEG — Negative , NNC — Compound Nouns , NNPC — Compound Proper Nouns , NVB — Noun in Kriyamula , JVB — Adjective in Kriyamula , RBVB — Adverb in Kriyamula , INF — Verb infinitival , QT — Quotative.

<u>LECTURE No.:12</u>

POS Tagging:

POS tagging is process of assigning a part-of-speech to each word in a corpus. Stochastic Tagging algorithm is implemented by the following models:

1. Hidden Markov Model (HMM)
2. Maximum Entropy Model (MEMM)
3. Conditional Random Fields (CRF)

The input to a tagging algorithm is a string of words and a specified tagset. The output is single best tag for each word.

Useful for many NLP works where a large volume of corpus is required to train this model which depicts the corpus-based POS tagging. Here the input is Untagged New Corpus and the output is Tagged New Corpus.

POS Tagged corpus

Learn

Untagged New Corpus --------→POS Tagger---------------→Tagged new corpus

Part-of-speech tagging

In corpus linguistics, part-of-speech tagging (POS tagging or PoS tagging or POST), also called grammatical tagging or word-category disambiguation, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context—i.e., its relationship with adjacent and related words in a phrase, sentence, or paragraph. A simplified form of this is commonly taught to school-age children, in the identification of words as nouns, verbs, adjectives, adverbs, etc. PoS tagging is used in natural language processing (NLP) and natural language understanding (NLU).

Example:

Word: Paper, Tag: Noun
Word: Go, Tag: Verb
Word: Famous, Tag: Adjective

The **part of speech** explains how a word is used in a sentence. There are eight main parts of speech - **nouns, pronouns, adjectives, verbs, adverbs, prepositions, conjunctions** and **interjections**.

Noun
Interjection
Pronoun
Parts of Speech
Preposition
Verb
Conjunction
Adverb
Adjective

- Noun (N)- Daniel, London, table, dog, teacher, pen, city, happiness, hope
- Verb (V)- go, speak, run, eat, play, live, walk, have, like, are, is
- Adjective(ADJ)- big, happy, green, young, fun, crazy, three
- Adverb(ADV)- slowly, quietly, very, always, never, too, well, tomorrow
- Preposition (P)- at, on, in, from, with, near, between, about, under
- Conjunction (CON)- and, or, but, because, so, yet, unless, since, if
- Pronoun(PRO)- I, you, we, they, he, she, it, me, us, them, him, her, this
- Interjection (INT)- Ouch! Wow! Great! Help! Oh! Hey! Hi!

Most **POS** are divided into sub-classes. **POS Tagging** simply means labeling words with their appropriate Part-Of-Speech. Once performed by hand, POS tagging is now done in the context of computational linguistics, using algorithms which associate discrete terms, as well as hidden parts of speech, in accordance with a set of descriptive tags. POS-tagging algorithms fall into two distinctive groups: rule-based and stochastic. E. Brill's tagger, one of the first and most widely used English POS-taggers, employs rule-based algorithms.

Principle of POST: Part-of-speech tagging is harder than just having a list of words and their parts of speech, because some words can represent more than one part of speech at different times, and because some parts of speech are complex or unspoken. This is not rare—in natural languages (as opposed to many artificial languages), a large percentage of word-forms are ambiguous.

A part-of-speech (PoS) tagger is a software tool that labels words as one of several categories to identify the word's function in a given language. In the English language, words fall into one of eight or nine parts of speech. Part-of-speech categories include noun, verb, article, adjective, preposition, pronoun, adverb, conjunction and interjection. PoS taggers use algorithms to label terms in text bodies. These taggers make more complex categories than those defined as basic PoS, with tags such as "noun-plural" or even more complex labels. Part-of-speech categorization is taught to school-age children in English grammar, where children perform basic PoS tagging as part of their education.

The process of assigning one of the parts of speech to the given word is called Parts Of Speech tagging. It is commonly referred to as POS tagging. Parts of speech include nouns, verbs, adverbs, adjectives, pronouns, conjunction and their sub-categories.

Parts Of Speech tagger: Parts Of Speech tagger or POS tagger is a program that does this job. Taggers use several kinds of information: dictionaries, lexicons, rules, and so on. Dictionaries have category or categories of a particular word. That is a word may belong to more than one category. For example, run is both noun and verb. Taggers use probabilistic information to solve this ambiguity.

There are mainly two type of taggers: rule-based and stochastic. Rule-based taggers use hand-written rules to distinguish the tag ambiguity. Stochastic taggers are either HMM based, choosing the tag sequence which maximizes the product of word likelihood and tag sequence probability, or cue-based, using decision trees or maximum entropy models to combine probabilistic features.

Ideally a typical tagger should be robust, efficient, accurate, tunable and reusable. In reality taggers either definitely identify the tag for the given word or make the best guess based on the available information. As the natural language is complex it is sometimes difficult for the taggers to make accurate decisions about tags. So, occasional errors in tagging are not taken as a major roadblock to research.

LECTURE No.:13

Architecture of POS tagger:

1. **Tokenization:** The given text is divided into tokens so that they can be used for further analysis. The tokens may be words, punctuation marks, and utterance boundaries.
2. **Ambiguity look-up:** This is to use lexicon and a guessor for unknown words. While lexicon provides list of word forms and their likely parts of speech, guessors analyze unknown tokens. Compiler or interpreter, lexicon and guessor make what is known as lexical analyzer.
3. **Ambiguity Resolution:** This is also called disambiguation. Disambiguation is based on information about word such as the probability of the word. For example, power is more likely used as noun than as verb. Disambiguation is also based on contextual information or word/tag sequences. For example, the model might prefer noun analyses over verb analyses if the preceding word is a preposition or article. Disambiguation is the most difficult problem in tagging.

PoS taggers categorize terms in PoS types by their relational position in a phrase, relationship with nearby terms and by the word's definition. PoS taggers fall into those that use stochastic methods, those based on probability and those which are rule-based. One of the first PoS taggers developed was the E. Brill tagger, a rule-based tagging tool. E. Brill is still commonly used today. Other tools that perform PoS tagging include Stanford Log-linear Part-Of-Speech Tagger, Tree Tagger, and Microsoft's POS Tagger. Part-of-speech tagging is also referred to as word category disambiguation or grammatical tagging.

Different POS Tagging Techniques: There are different techniques for POS Tagging:

1. **Lexical Based Methods—A**ssigns the POS tag the most frequently occurring with a word in the training corpus.

2. **Rule-Based Methods—**Assigns POS tags based on rules. For example, we can have a rule that says, words ending with "ed" or "ing" must be assigned to a verb. Rule-Based Techniques can be used along with Lexical Based approaches to allow POS Tagging of words that are not present in the training corpus but are there in the testing data.

3. **Probabilistic Methods—**This method assigns the POS tags based on the probability of a particular tag sequence occurring. Conditional Random Fields (CRFs) and Hidden Markov Models (HMMs) are probabilistic approaches to assign a POS Tag.

4. **Deep Learning Methods**—Recurrent Neural Networks can also be used for POS tagging.

Conditional Random Fields(CRF): A CRF is a **Discriminative Probabilistic Classifiers**. The difference between discriminative and generative models is that while **discriminative models try to model conditional probability distribution, i.e., P(y|x), generative models try to model a joint probability distribution, i.e., P(x,y)**. *Logistic Regression, SVM, CRF are Discriminative Classifiers. Naive Bayes, HMMs are Generative Classifiers.* CRF's can also be used for sequence labelling tasks like Named Entity Recognisers and POS Taggers.
In CRFs, the input is a set of features (real numbers) derived from the input sequence using feature functions, the weights associated with the features (that are learned) and the previous label and the task is to predict the current label. The weights of different feature functions will be determined such that the likelihood of the labels in the training data will be maximised.
In CRF, a set of feature functions are defined to extract features for each word in a sentence. Some examples of feature functions are: is the first letter of the word capitalised, what the suffix and prefix of the word, what is the previous word, is it the first or the last word of the sentence, is it a number etc. These set of features are called **State Features.** In CRF, we also pass the label of the previous word and the label of the current word to learn the weights. CRF will try to determine the weights of different feature functions that will maximise the likelihood of the labels in the training data. The feature function dependent on the label of the previous word is **Transition Feature.**

Tagset: Tagset is the set of tags from which the tagger is supposed to choose to attach to the relevant word.
Every tagger will be given a standard tagset. The tagset may be coarse such as N (Noun), V(Verb), ADJ(Adjective), ADV(Adverb), PREP(Preposition), CONJ(Conjunction) or fine-grained such as NNOM(Noun-Nominative), NSOC(Noun-Sociative), VFIN(Verb Finite),VNFIN(Verb Nonfinite) and so on. Most of the taggers use only fine grained tagset.

Implementation:

- Correct grammatical tagging will reflect that "dogs" is here used as a verb, not as the more common plural noun. Grammatical context is one way to determine this; semantic analysis can also be used to infer that "sailor" and "hatch" implicate "dogs" as 1) in the nautical context and 2) an action applied to the object "hatch" (in this context, "dogs" is a nautical term meaning "fastens (a watertight door) securely").

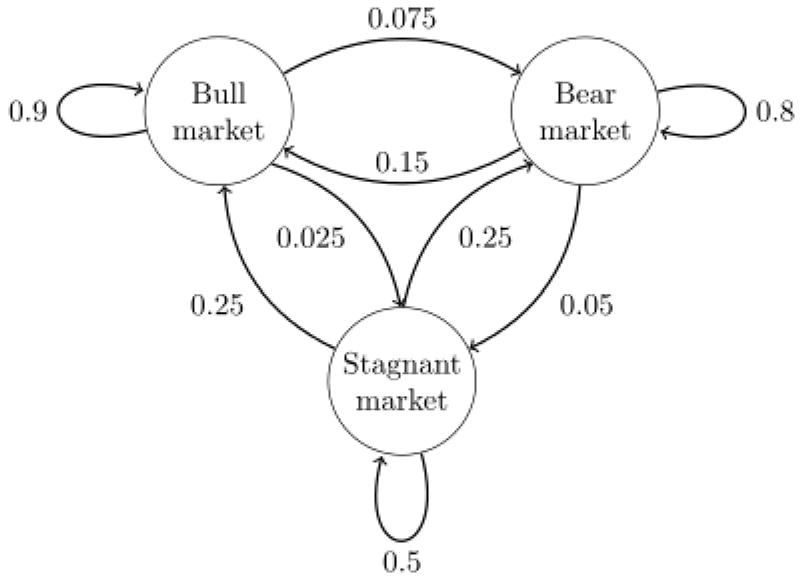
- Schools commonly teach that there are 9 parts of speech in English: noun, verb, article, adjective, preposition, pronoun, adverb, conjunction, and interjection. However, there are clearly many more categories and sub-categories. For nouns, the plural, possessive, and singular forms can be distinguished. In many languages words are also marked for their "case" (role as subject, object, etc.), grammatical gender, and so on; while verbs are marked for tense, aspect, and other things. Linguists distinguish parts of speech to various fine degrees, reflecting a chosen "tagging system".
- In part-of-speech tagging by computer, it is typical to distinguish from 50 to 150 separate parts of speech for English. For example, NN for singular common nouns, NNS for plural common nouns, NP for singular proper nouns (see the POS tags used in the Brown Corpus). Work on stochastic methods for tagging Koine Greek (DeRose 1990) has used over 1,000 parts of speech, and found that about as many words were ambiguous there as in English. A morphosyntactic descriptor in the case of morphologically rich languages is commonly expressed using very short mnemonics, such as Ncmsan for Category=Noun, Type = common, Gender = masculine, Number = singular, Case = accusative, Animate = no.

LECTURE No.:14

Markov chain:

A Markov chain is a stochastic process with the Markov property. The term "Markov chain" refers to the sequence of random variables such a process moves through, with the Markov property defining serial dependence only between adjacent periods (as in a "chain").[1] It can thus be used for describing systems that follow a chain of linked events, where what happens next depends only on the current state of the system.

A Markov chain essentially consists of a set of transitions, which are determined by some probability distribution, that satisfy the ***Markov property***. Formally, a Markov chain is a probabilistic automaton. The probability distribution of state transitions is typically represented as the Markov chain's ***transition matrix***. If the Markov chain has **N** possible states, the matrix will be an **N x N** matrix, such that entry **(I, J)** is the probability of transitioning from state **I** to state **J**. Additionally, the transition matrix must be a **stochastic matrix**, a matrix whose entries in each row must add up to exactly 1. This makes complete sense, since each row represents its own probability distribution.



General view of a sample Markov chain, with states as circles, and edges as transitions

$$P = \begin{bmatrix} 0.9 & 0.075 & 0.025 \\ 0.15 & 0.8 & 0.05 \\ 0.25 & 0.25 & 0.5 \end{bmatrix}.$$

Sample transition matrix with 3 possible states

Additionally, a Markov chain also has an ***initial state vector***, represented as an **N x 1** matrix (a vector), that describes the probability distribution of starting at each of the **N** possible states. Entry **I** of the vector describes the probability of the chain beginning at state **I**.

Hidden Markov models

Markov model assumes that each state can be uniquely associated with an observable event. Once an observation is made, the state of the system is then trivially retrieved. This model, however, is too restrictive to be of practical use for most realistic problems–to make the model more flexible, we will assume that the outcomes or observations of the model are a probabilistic function of each state. Each state can produce a number of outputs according to a unique probability distribution, and each distinct output can potentially be generated at any state. These are known a Hidden Markov Models(HMM), because the state sequence is not directly observable, it can only be approximated from the sequence of observations produced by the system

Set of states: $\{ s_1,s_2,\ldots\ldots,s_n\}$
Process moves from one state to another generating a sequence of states : $\{s_{i1},s_{i2},$
$\ldots,s_{ik},\ldots\ldots\}$
Markov chain property: probability of each subsequent state depends only on what was the previous state:

$$P(S_{ik}|S_{i1},S_{i2},\ldots\ldots S_{ik-1})=P(S_{ik}|S_{ik-1})$$

States are not visible, but each state randomly generates one of M observations (or visible states) $\{v_1,v_2,\ldots\ldots,v_M\}$
To define hidden Markov model, the following probabilities have to be specified: matrix of transition probabilities $A=(a_{ij})$, $a_{ij}= P(s_i| s_j)$, matrix of observation probabilities B=(bi (vm)), bi(vm)= P(vm| si) and a vector of initial probabilities π=(πi), πi = P(si) .
Model is represented by M=(A, B, π).€$P(s_{ik}|s_{i1},s_{i2},\ldots,s_{ik-1})=P(s_{ik}|s_{ik-1})$

Example of Hidden Markov Model

Two states : ‘Low’ and ‘High’ atmospheric pressure.
Two observations : ‘Rain’ and ‘Dry’.

Transition probabilities: P('Low'|'Low')=0.3 , P('High'|'Low')=0.7 , P('Low'|'High')=0.2, P('High'|'High')=0.8
Observation probabilities : P('Rain'|'Low')=0.6 , P('Dry'|'Low')=0.4 , P('Rain'|'High')=0.4 , P('Dry'|'High')=0.3 .
Initial probabilities: say P('Low')=0.4 , P('High')=0.6 .

Calculation of observation sequence probability

Suppose we want to calculate a probability of a sequence of observations in our example, {'Dry','Rain'}.
Consider all possible hidden state sequences:
P({'Dry','Rain'}) = P({'Dry','Rain'} , {'Low','Low'}) + P({'Dry','Rain'} , {'Low','High'}) + P({'Dry','Rain'} , {'High','Low'}) + P({'Dry','Rain'} , {'High','High'}) where first term is : P({'Dry','Rain'} , {'Low','Low'})= P({'Dry','Rain'} | {'Low','Low'}) P({'Low','Low'})
= P('Dry'|'Low')P('Rain'|'Low') P('Low')P('Low'|'Low)
= 0.4*0.4*0.6*0.4*0.3

Forward and Backward procedures

Probability Evaluation–Our goal is to compute the likelihood of an observation sequence $O=o1,o2,o3...$given a particular HMM model $\lambda= \{A,B,\pi\}$–Computation of this probability involves enumerating every possible state sequence and evaluating the corresponding probability$P(O|\lambda)=\Sigma_{\forall Q}\ P(O|Q,\lambda)P(Q|\lambda)$
For a particular state sequence $Q=\{q1,q2,q3,...,\}$ $P(O|Q,\lambda)$ is
$P(O|Q,\lambda)=\Pi_{t=1}^{T}\ P(ot|qt,\lambda)= \Pi_{t=1}^{T}\ b_{qt}(ot)$
The probability of the state sequence $Q$is $P(Q|\lambda)=\pi_{q1}a_{q1q2}a_{q2q3}...a_{qT-1qT}$

LECTURE No.:15

Viterbi algorithm:

The **Viterbi algorithm** is a dynamic programming algorithm for finding the most likely sequence of hidden states—called the **Viterbi path**—that results in a sequence of observed events, especially in the context of Markov information sources and hidden Markov models.

This algorithm generates a path X= $(x_1,x_2,\ldots\ldots,x_T)$ which is a sequence of states

X_n € S = $\{s_1,s_2,\ldots..,s_k\}$ that generate the observations Y = $(y_1,y_2,\ldots.,y_T)$with

y_n € O = $\{o_1,o_2,\ldots.,o_N\}$(Nbeing the count of observations (observation space, see below)).

Two 2-dimensional tables of size K x T are constructed:

Each element $T_1[i,j]$ of T_1 stores the probability of the most likely path so far
X' = $(x_1',x_2',\ldots.,x_j')$ with $x_j' = s_i$ that generates Y = $(y_1,y_2,\ldots.,y_j)$.
Each element $T_2[i,j]$ of T_2 stores x_{j-1}' of the most likely path so far
X'= $(x_1',x_2',\ldots.,x_j' = s_i)$ for ˅ j, 2≤ j ≤ T
The table entries $T_1[i,j]$, $T_2[i,j]$ are filled by increasing order of K . j + i
$T_1[i,j] = \max_k (T_1[k,j-1].A_{ki}.B_{iy_j})$ and $T_2[i,j] = \text{argmax}_k (T_1[k,j-1].A_{ki})$ with A_{ki} and B_{iy_j} are defined below:
B_{iy_j} does not need to appear in the latter expression, as it's non-negative and independent of k and thus does not affect the argmax.
INPUT

- The observation space O = $\{o_1,o_2,\ldots.,o_N\}$
- the state space S = $\{s_1,s_2,\ldots..,s_k\}$
- an array of initial probabilities Π = { Π_1 , Π_2 ,….., Π_k } such that Π_I stores the probability that $x_1==s_i$
- A sequence of observations Y = $\{y_1,y_2,\ldots.,y_k\}$ such that y_t==i if the observation at time t is o_i
- transition matrix A of size K x K such that A_{ij} stores the transition probability of transiting from state $s_i$to state s_j
- emission matrix B of size K x N such that B_{ij} stores the probability of observing o_j from state s_i

OUTPUT: The most likely hidden state sequence X= $(x_1,x_2,\ldots\ldots,x_T)$

Word recognition example:

Hidden states of HMM = characters
• Observations = typed images of characters segmented from the image . There is an infinite number of observations
• Observation probabilities = character recognizer scores. •Transition probabilities will be defined differently in two subsequent models.

If lexicon is given, we can construct separate HMM models for each lexicon word.

Here recognition of word image is equivalent to the problem of evaluating few HMM models.
This is an application of Evaluation problem.
We can construct a single HMM for all words.
Hidden states = all characters in the alphabet.
Transition probabilities and initial probabilities are calculated from language model.

MODULE 1: REGULAR EXPRESSION AND AUTOMATA

LECTURE 1: INTREODUCTION TO FINITE AUTOMATA

FINITE AUTOMATA:

A finite automaton can be represented as a 5-tuple structure:

$M = (Q, \Sigma, \delta, q_0, F)$

where,

Q is set of states of the system

Σ is input alphabet of the system

δ is transition function of the system defined as $\delta : Q \times \Sigma \rightarrow Q$

q_0 is the initial or start state of the system

F is the set of final states of the system

Finite Automata are of two types: Non-deterministic Finite Automata (NFA/NDFA) and Deterministic Finite Automata (DFA).

1.1. NON-DETERMINISTIC FINITE AUTOMATA:

A FA is said to be non-deterministic if for a particular input is applied to a current state it is not sure about the next state.

A non-deterministic finite automaton can be represented as a 5-tuple structure:

$M = (Q, \Sigma, \delta, q_0, F)$

where,

Q is set of states of the finite automata

Σ is input alphabet of the finite automata

δ is transition function of the finite automata defined as $\delta : Q \times \Sigma \rightarrow 2^Q$

q_0 is the initial or start state of the finite automata

F is the set of final states of the finite automata

A non-deterministic finite automaton can be denoted by transition diagram or by transition table or by transition function.

Let us take an example of a non-deterministic finite automaton

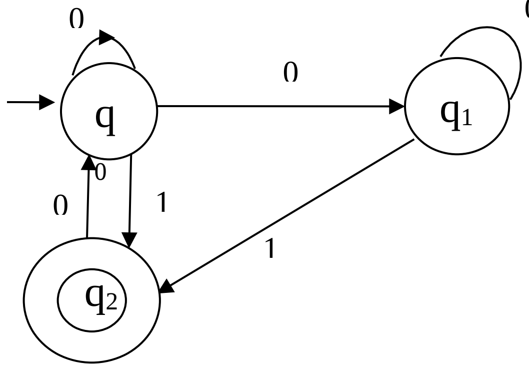


Figure 1.1: Transition Diagram of NFA M1

We can represent the above NFA by a transition table as follows:

PS	**NS**	
	x=0	**x=1**
→q0	{q0,q1}	{q2}
q1	{q1}	{q2}
*q2	{q0}	-

* Final state

We can represent NFA M1 by 5-tuple structure as follows:

Q = {q0,q1,q2}
Σ = {0,1}
δ(q0,0)={q0,q1}, δ(q0,1)={q2}, δ(q1,0)={q1}, δ(q1,1)={q2}, δ(q2,0)={q0}, δ(q2,1)=Φ
q_0 = q0
F={q2}

1.2. DETERMINISTIC FINITE AUTOMATA:

A FA is said to be non-deterministic if for a particular input is applied to a current state it is very sure about the next state.

A deterministic finite automaton can be represented as a 5-tuple structure:

M = (Q, Σ, δ, q_0, F)

where,

Q is set of states of the finite automata
Σ is input alphabet of the finite automata
δ is transition function of the finite automata defined as $\delta : Q \times \Sigma \rightarrow Q$
q_0 is the initial or start state of the finite automata
F is the set of final states of the finite automata

Let us take an example of a deterministic finite automaton

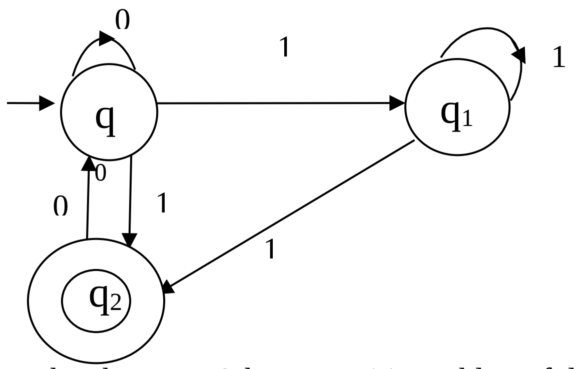


Figure 1.2: Transition Diagram of DFA M2

We can represent the above DFA by a transition table as follows:

PS	**NS**	
	x=0	**x=1**
→q0	{q0}	{q1}
q1	{q1}	{q2}
*q2	{q0}	-

* Final state

We can represent DFA M2 by 5-tuple structure as follows:

Q = {q0,q1,q2}
Σ = {0,1}
δ(q0,0)={q0}, δ(q0,1)={q1}, δ(q1,0)={q1}, δ(q1,1)={q2}, δ(q2,0)={q0}, δ(q2,1)=Φ
q_0 = q0
F={q2}

LECTURE 2: CONTEXT FREE GRAMMAR

DEFINITION:

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

- **N** is a set of non-terminal symbols.
- **T** is a set of terminals where **N ∩ T =NULL.**
- **P** is a set of rules, **P: N → (N U T)***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
- **S** is the start symbol.

EXAMPLE:

1. The grammar ({A}, {a, b, c}, P, A), P : A → aA, A →abc.
2. The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S →ε
3. The grammar ({S, F}, {0, 1}, P, S), P: S → 00S|11F, F → 00F |ε

DERIVATION TREE:

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

REPRESENTATION TECHNIQUE:

i. **Root vertex:** Must be labeled by the start symbol.
ii. **Vertex:** Labeled by a non-terminal symbol.
iii. **Leaves:** Labeled by a terminal symbol or ε.

EXAMPLE:
If S → x_1x_2 …… xn is a production rule in a CFG, then the parse tree / derivation tree will be as follows:

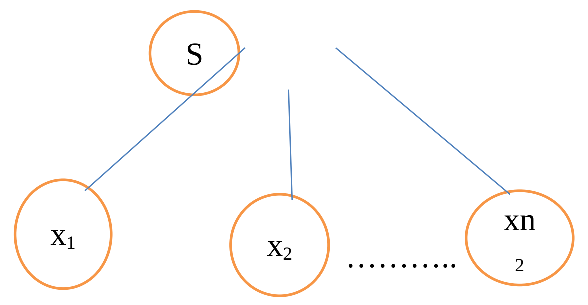


FIG3.1: Derivation Tree for the production $S \rightarrow x_1x_2 \ldots\ldots xn$

DERIVATION OR YIELD OF A PARSE TREE:
The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

Example
Let a CFG {N,T,P,S} be
N = {S}, T = {a, b}, Starting symbol = S, P = $S \rightarrow SS \mid aSb \mid \varepsilon$
One derivation from the above CFG is “abaabb”
$S \rightarrow SS \rightarrow aSbS \rightarrow abS \rightarrow abaSb \rightarrow abaaSbb \rightarrow abaabb$

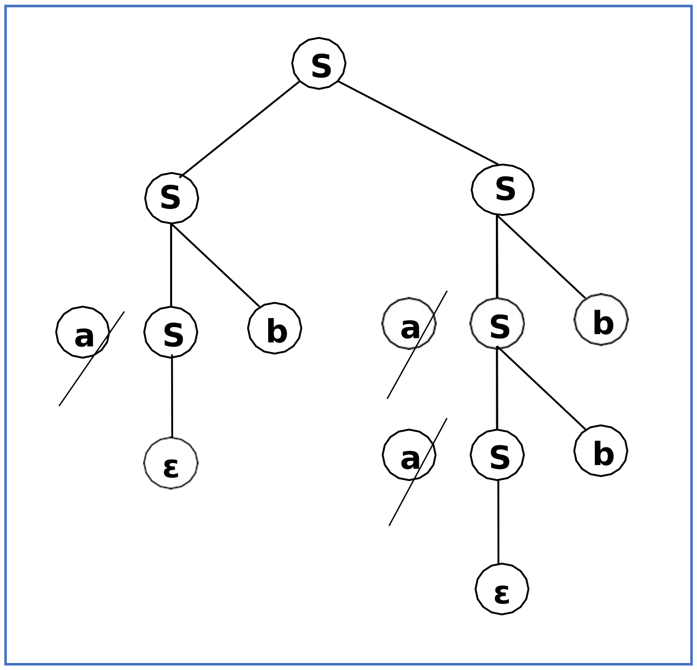
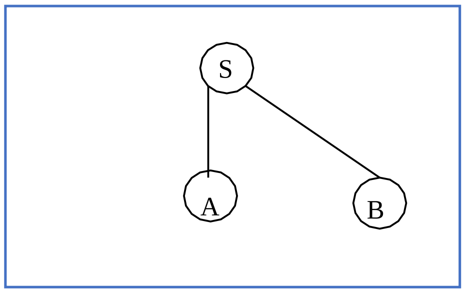


FIG 3. : Sentential Form and Partial Derivation Tree

A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

Example
If in any CFG the productions are:
S→AB, A → aaA | ε, B →Bb| ε the partial derivation tree can be the following:



If a partial derivation tree contains the root **S**, it is called a **sentential form**. The above sub-tree is also in sentential form.

Leftmost and Rightmost Derivation of a String
Leftmost derivation - A leftmost derivation is obtained by applying production to the leftmost variable in each step.

Rightmost derivation - A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example
Let any set of production rules in a CFG be X → X+X | X*X |X| a
over an alphabet {a}.
The leftmost derivation for the string "**a+a*a**" may be:
X → X+X→ a+X→ a+ X*X →a+a*X→a+a*a
The stepwise derivation of the above string is shown as below

The right most derivation for the above string "**a+a*a**" may be:

X → X*X→ X*a → X+X*a →X+a*a→a+a*a

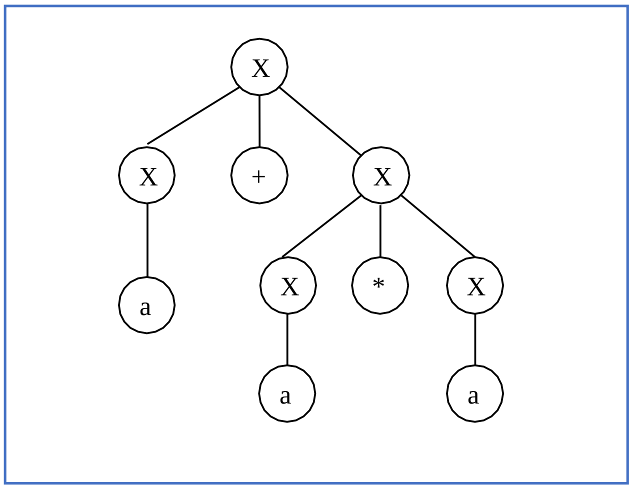
LEFT AND RIGHT RECURSIVE GRAMMAR:

In a context-free grammar G, if there is a production in the form X → Xa where X is a non-terminal and 'a' is a string of terminals, it is called a left recursive production. The grammar having a left recursive production is called a left recursive grammar.
And if in a context-free grammar G, if there is a production is in the form X → aX where X is a non-terminal and 'a' is a string of terminals, it is called a right recursive production. The grammar having a right recursive production is called a right recursive grammar.
IfacontextfreegrammarGhasmorethanonederivationtreeforsomestringw∈L(G), it is called an ambiguous grammar. There exist multiple right-most or left-most derivations for some string generated from that grammar.

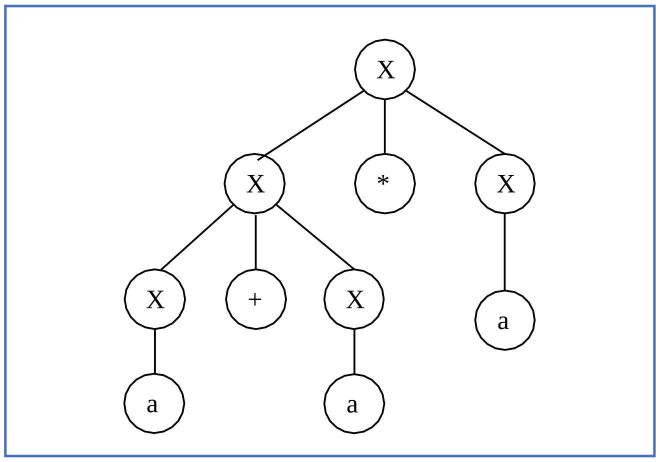
PROBLEM

Check whether the grammar **G** with production rules:
X → X+X | X*X |X| a is ambiguous or not.
Solution
Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

Derivation 1: X → X+X→ a +X→ a+ X*X →a+a*X→ a+a*a
PARSE TREE 1:



Derivation2: X → X*X→X+X*X→ a+ X*X →a+a*X→a+a*a
PARSE TREE 2:



Since there are two parse trees for a single string "a+a*a", the grammar **G** is ambiguous.

LECTURE 3: WORD TOKENIZATION

Text Normalization

Before almost any natural language processing of a text, the text has to be normalized. At least three tasks are commonly applied as part of any normalization process:

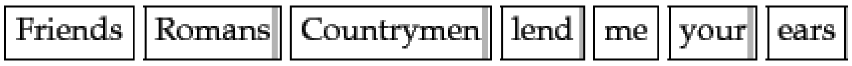
1. **Segmenting/tokenizing words from running text**
2. **Normalizing word formats**
3. **Segmenting sentences in running text.**

In the next sections we walk through each of these tasks.

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 sometimes important to make a type/token distinction. A *token* is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing. A *type* is the class of all tokens containing the same character sequence. A *term* is a (perhaps normalized) 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 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 being exactly the tokens that appear in the document, they are usually derived from them by various normalization processes . 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 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?
$\boxed{\text{neill}}$
$\boxed{\text{oneill}}$
$\boxed{\text{o'neill}}$
$\boxed{\text{o'}}$ $\boxed{\text{neill}}$
$\boxed{\text{o}}$ $\boxed{\text{neill}}$?
And for *aren't*, is it:
$\boxed{\text{aren't}}$
$\boxed{\text{arent}}$
$\boxed{\text{are}}$ $\boxed{\text{n't}}$
$\boxed{\text{aren}}$ $\boxed{\text{t}}$?
A simple strategy is to just split on all non-alphanumeric characters, but while $\boxed{\text{o}}$ $\boxed{\text{neill}}$ looks okay, $\boxed{\text{aren}}$ $\boxed{\text{t}}$ 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 Boolean or free text queries, you always want to do the exact same tokenization 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.

These issues of tokenization are language-specific. It thus requires the language of the document to be known. *Language identification* based on classifiers that use short character subsequences as features is highly effective; most languages have distinctive signature patterns. 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 introduced 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, numbers, and URLs, since their presence greatly expands the size of the vocabulary. 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* parametricsection.

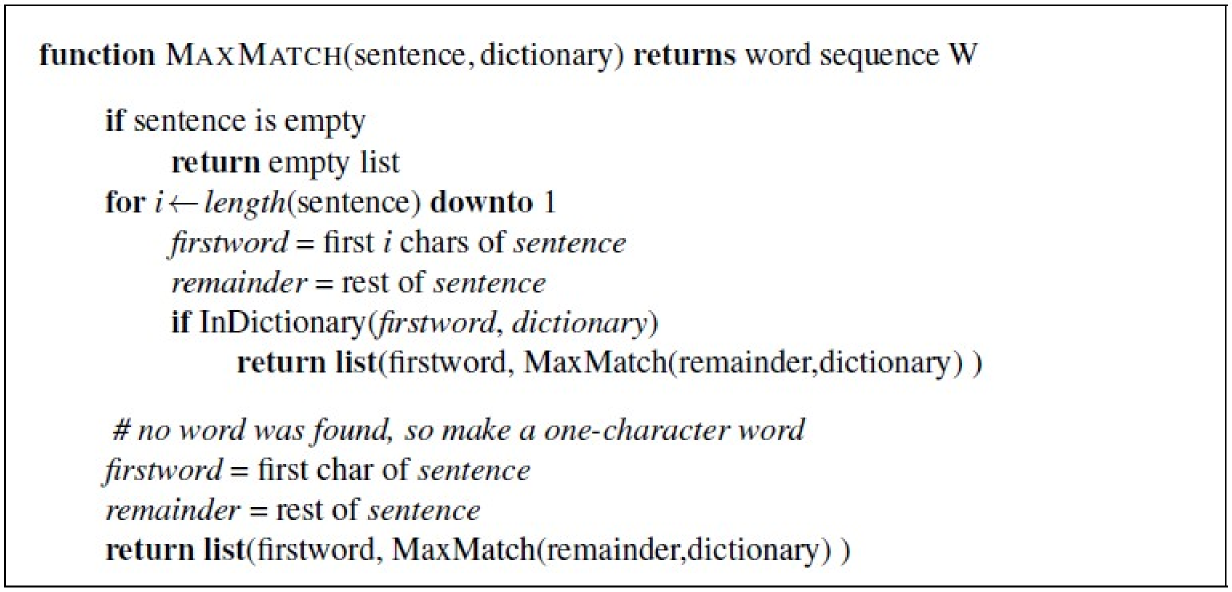
In English, *hyphenation* is used for various purposes ranging from splitting up vowels in words (*co-education*) to joining nouns as names (*Hewlett-Packard*) to a copyediting device to show word grouping (*the hold-him-back-and-drag-him-away maneuver*). It is easy to feel that the first example should be regarded as one token (and is indeed more commonly written as just *coeducation*), 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 regarded as a single token. This occurs most commonly with names (*San Francisco, Los Angeles*) but also with borrowed foreign phrases (*au fait*) and compounds 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, 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 handled 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 (westlaw), is to encourage users to enter hyphens wherever they may be possible, and whenever there is a hyphenated form, the system will generalize 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 *compound nouns* without spaces (e.g., *Computerlinguistik* `computational linguistics'; *Lebensversicherungsgesellschaftsangestellter* `life insurance company employee'). Retrieval systems for German greatly benefit from the use of a *compound-splitter* module, which is usually implemented by seeing if a word can be subdivided into multiple words that appear in a vocabulary. This phenomenon reaches its limit case with major East Asian Languages (e.g., Chinese, Japanese, Korean, and Thai), where text is written without any spaces between words. One approach here is to 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 conditional random fields, trained over hand-segmented words. Since there are multiple possible segmentations of character sequences, all such methods make mistakes sometimes, and so you are never guaranteed a consistent unique tokenization. The other approach is to abandon word-based indexing and to do all indexing via just short subsequences of characters (character n -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.

Word Segmentation in Chinese: the MaxMatch algorithm

Some languages, including written Chinese, Japanese, and Thai, do not use spaces to mark potential word-boundaries, and so require alternative segmentation methods. In Chinese, for example, words are composed of characters known as **hanzi**. Each character generally represents a single morpheme and is pronounceable as a single syllable. Words are about 2.4 characters long on average. A simple algorithm that does remarkably well for segmenting Chinese, and often used as a baseline comparison for more advanced methods, is a version of greedy search called **maximum matching** or sometimes **MaxMatch**. The algorithm requires a dictionary (wordlist) of the language.
The maximum matching algorithm starts by pointing at the beginning of a string. It chooses the longest word in the dictionary that matches the input at the current position. The pointer is then advanced to the end of that word in the string. If no word matches, the pointer is instead advanced one character (creating a one- character word). The algorithm is then iteratively applied again starting from the new pointer position. Fig shows a version of the algorithm.



```
function MAXMATCH(sentence, dictionary) returns word sequence W

    if sentence is empty
        return empty list
    for i ← length(sentence) downto 1
        firstword = first i chars of sentence
        remainder = rest of sentence
        if InDictionary(firstword, dictionary)
            return list(firstword, MaxMatch(remainder,dictionary) )

    # no word was found, so make a one-character word
    firstword = first char of sentence
    remainder = rest of sentence
    return list(firstword, MaxMatch(remainder,dictionary) )
```

Fig: The MaxMatch algorithm for word segmentation.

MaxMatch doesn’t work as well on English. To make the intuition clear, we’ll create an example by removing the spaces from the beginning of Turing’s famous quote “We can only see a short distance ahead”, producing “wecanonlyseeashortdis- tanceahead”. The MaxMatch results are shown below.

Input: wecanonlyseeashortdistanceahead

Output: we canon l y see ash ort distance ahead

On English the algorithm incorrectly chose canon instead of stopping at can, which left the algorithm confused and having to create single-character words like l and y.

Sentence Segmentation

Sentence segmentation is another important step in text processing. The most use- ful cues for segmenting a text into sentences are punctuation, like periods, question marks, exclamation points. Question marks and exclamation points are relatively unambiguous markers of sentence boundaries. Periods, on the other hand, are more ambiguous. The period character “.” is ambiguous between a sentence boundary marker and a marker of abbreviations like *Mr.* or *Inc.* The previous sentence that you just read showed an even more complex case of this ambiguity, in which the final period of *Inc.* marked both an abbreviation and the sentence boundary marker. For this reason, sentence tokenization and word tokenization may be addressed jointly.

In general, sentence tokenization methods work by building a binary classifier (based on a sequence of rules or on machine learning) that decides if a period is part of the word or is a sentence-boundary marker. In making this decision, it helps to know if the period is attached to a commonly used abbreviation; thus, an abbreviation dictionary is useful.

State-of-the-art methods for sentence tokenization are based on machine learning and are introduced in later chapters.

Computational Lexical Semantics

1. Introduction to Lexical Semantics

The branch of semantics that deals with the word meaning is called lexical semantics. It is the study of systematic, meaning related structures of words. Lexical semantics examines relationships among word meanings. It is the study of how the lexicon is organized and how the lexical meanings of lexical items are interrelated, and it's principal goal is to build a model for the structure of the lexicon by categorizing the types of relationships between words. Hyponymy , homonymy, polysemy, synonymy, antonymy and metonymy are different types of lexical relations.

1.1 Homonymy:

The word Homonym has been derived from Greek term 'Homoios' which means identical and 'onoma' means name. So, Homonymy is a relation that holds between two lexemes that have the same form but unrelated meanings. Homonyms are the words that have same phonetic form (homophones) or orthographic form (homographs) but different unrelated meanings. The ambiguous word whose different senses are far apart from each other and are not obviously related to each other in any way is called as Homonymy. Words like tale and tail are homonyms. There is no conceptual connection between its two meanings. For example the word 'bear', as a verb means 'to carry' and as a noun it means 'large animal'. An example of homonym which is both homophone and homograph is the word 'fluke'. Fluke is a fish as well as a flatworm. Other examples are bank, an anchor, and so on.

Homophony - Homophony is the case where two words are pronounced identically but they have different written forms. They sound alike but are written differently and often have different meanings. For example: no-know, led-lead, would-wood.

Homograph - Homograph is a word which is spelled the same as another word and might be pronounced the same or differently but which has a different. For example, Bear-bear ; Read-read.

When homonyms are spelled the same they are homographs but not all homonyms are homographs.

1.2 Hyponymy:

Hyponymy is a sense relation in semantics that serves to relate word concepts in a hierarchical fashion. Hyponymy is a relation between two words in which the meaning of one of the words includes the meaning of the other word. The lexical relation corresponding to the inclusion of one

class in another is hyponymy. Examples are : apple- fruit ; car- vehicles ; tool- furntiture ; cow - animal. The more specific concept is known as the hyponym, and the more general concept is known as the hypernym or superordinate. Apple is the hyponym and fruit is the superordinate / hypernymy. Hyponymy is not restricted to objects, abstract concepts, or nouns. It can be identified in many other areas of the lexicon.
E.g :
a. the verb cook has many hyponyms.
Word: Cook
Hyponyms: Roast, boil, fry, grill, bake.
b. the verb colour has many hyponyms

Word: colour
Hyponyms: blue, red, yellow, green, black and purple
Hyponymy involves the logical relationship of entailment. Example : ‘There is a horse’ entails that ‘There is an animal”. Hyponymy often functions In discourse as a means of lexical cohesion by establishing referential equivalence to avoid repetition.

1.3 Polysemy:
A polyseme the phenomenon of having or being open to several or many meanings. When a word has several very closely related senses or meanings. Polysemous word is a word having two or more meanings. For example, foot in : - He hurt his foot ; - She stood at the foot of the stairs.
A well-known problem in semantics is how to decide whether we are dealing with a single polysemous word or with two or more homonyms.
F.R.Palmer concluded saying that finally multiplicity of meaning is a very general characteristic of language. Polysemy is used in semantics and lexical analysis to describe the word with multiple meanings. Crystal and Dick Hebdige (1979) also defined polysemy. Lexical ambiguity depends upon homonymy and polysemy.
The difference between homonyms and polysemes is subtle. Lexicographers define polysemes within a single dictionary lemma, numbering different meanings, while homonyms are treated in separate lemmata. Semantic shift can separate a polysemous word into separate homonyms. For example, check as in "bank check" (or Cheque) , check in chess, and check meaning "verification" are considered homonyms, while they originated as a single word derived from chess in the 14th century.

1.4 Synonymy:
The semantic qualities or sense relations that exist between words (lexemes) with closely related meanings (i.e., synonyms). Plural: synonymies. Contrast with antonymy. Synonymy may also refer to the study of synonyms or to a list of synonyms. In the words of Dagmar Divjak, near-synonymy (the relationship between different lexemes that express similar meanings) is "a fundamental phenomenon that influences the structure of our lexical knowledge" (Structuring the Lexicon, 2010). Words that are synonyms are said to be synonymous, and the state of being a synonym is called synonymy. For example, the words begin, start, commence, and initiate are all synonyms of one another. Words are typically synonymous in one particular sense: for example, long and extended in the context long time or extended time are synonymous, but long cannot be used in the phrase extended family. Synonyms with the exact same meaning share a seme or denotational sememe, whereas those with inexactly similar meanings share a broader denotational or connotational sememe and thus overlap within a semantic field. The former are sometimes called cognitive synonyms and the latter, near-synonyms, plesionyms or poecilonyms.
Synonyms can be any part of speech, as long as both words belong to the same part of speech. Examples:

noun: drink and beverage
verb : buy and purchase
adjective : big and large
adverb :quickly and speedily
preposition :on and upon
Synonyms are defined with respect to certain senses of words: pupil as the aperture in the iris of the eye is not synonymous with student. Such like, he expired means the same as he died, yet my passport has expired cannot be replaced by my passport has died.

1.5 Thesaurus:

A thesaurus is a resource that groups words according to similarity. Thesauruses such as Roget and WordNet are produced manually, whereas others, as in pioneering work by Sparck Jones (1986) and more recent advances from Grefenstette (1994) and Lin (1998) are produced automatically from text corpora. One might consider the manually-produced ones to be semantic, since lexicographers put words in the same group according to their meaning, whereas the automatically produced ones are distributional, since the computer classifies them according to distribution. However there are both theoretical and practical arguments against viewing them as different sorts of objects.
The theoretical argument refers to Wittgenstein's “don't ask for the meaning, ask for the use” (1953). When invoking meaning as an organizing principle, we are invoking a highly problematic concept about which philosophers have argued since Plato, and they show no signs of stopping now. It is not clear what it means to say words in the same thesaurus cluster have similar meanings: the logician’s response that synonyms are words that can always be exchanged salve vertitate –without affecting the truth value of the sentence– tells us nothing about word senses, or about circumstances where one word is more apt or accurate then another, and probably implies there are no, or very few, synonyms. Justeson and Katz (1991) demonstrate how one supposedly semantic relation, antonymy, key to the mental lexicon for adjectives (Miller 1998), ceases to be mysterious exactly when it is re-interpreted as a distributional relation. To understand or evaluate any thesaurus, we would do well to consider the distributional as well as the semantic evidence.
The practical argument is simply that semantic and distributional thesauruses are both tools we might use for the same purposes. If we wish to know what thesaurus is best for a given task, both kinds are candidates and should be compared.
Some thesauruses, usually manual ones, have hierarchical structure involving a number of layers. Others, usually the automatic ones, simply comprise groups of words (so may be viewed as one-level hierarchies). Hierarchical clustering algorithms may be applied to automatic thesauruses to make them multi-level (though this is hard to do well). The more-than-one-level hierarchies produced by algorithm will generally be simple hierarchies. The hierarchies produced manually are not --which leads us on to the vexed question of word senses.

1.5.1 WordNet:

WordNet® is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. The resulting network of meaningfully related words and concepts can be navigated with the browser. WordNet is also freely and publicly available for download. WordNet's structure makes it a useful tool for computational linguistics and natural language processing. WordNet superficially resembles a thesaurus, in that it groups words together based on their meanings. However, there are some important distinctions. First, WordNet interlinks not just word forms—strings of letters—but specific senses of words. As a result, words that are found in close proximity to one another in the network are semantically disambiguated. Second, WordNet labels the semantic relations among words, whereas the groupings of words in a thesaurus does not follow any explicit pattern other than meaning similarity.

1.5.1.1 Structure:

The main relation among words in WordNet is synonymy, as between the words shut and close or car and automobile. Synonyms--words that denote the same concept and are interchangeable in many contexts--are grouped into unordered sets (synsets). Each of WordNet's 117 000 synsets is linked to other synsets by means of a small number of "conceptual relations." Additionally, a synset contains a brief definition ("gloss") and, in most cases, one or more short sentences illustrating the use of the synset members. Word forms with several distinct meanings are represented in as many distinct synsets. Thus, each form-meaning pair in WordNet is unique.

1.5.1.2 Relations:

The most frequently encoded relation among synsets is the super-subordinate relation (also called hyperonymy, hyponymy or ISA relation). It links more general synsets like {furniture, piece_of_furniture} to increasingly specific ones like {bed} and {bunkbed}. Thus, WordNet states that the category furniture includes bed, which in turn includes bunkbed; conversely, concepts like bed and bunkbed make up the category furniture. All noun hierarchies ultimately go up the root node {entity}. Hyponymy relation is transitive: if an armchair is a kind of chair, and if a chair is a kind of furniture, then an armchair is a kind of furniture. WordNet distinguishes among Types (common nouns) and Instances (specific persons, countries and geographic entities). Thus, armchair is a type of chair, Barack Obama is an instance of a president. Instances are always leaf (terminal) nodes in their hierarchies.

Meronymy, the part-whole relation holds between synsets like {chair} and {back, backrest}, {seat} and {leg}. Parts are inherited from their superordinates: if a chair has legs, then an armchair has legs as well. Parts are not inherited "upward" as they may be characteristic only of specific kinds of things rather than the class as a whole: chairs and kinds of chairs have legs, but not all kinds of furniture have legs.

Verb synsets are arranged into hierarchies as well; verbs towards the bottom of the trees (troponyms) express increasingly specific manners characterizing an event, as in {communicate}-{talk}-{whisper}. The specific manner expressed depends on the semantic field; volume (as in the example above) is just one dimension along which verbs can be elaborated. Others are speed (move-jog-run) or intensity of emotion (like-love-idolize). Verbs describing events that necessarily and unidirectionally entail one another are linked: {buy}-{pay}, {succeed}-{try}, {show}-{see}, etc.

Adjectives are organized in terms of antonymy. Pairs of "direct" antonyms like wet-dry and young-old reflect the strong semantic contract of their members. Each of these polar adjectives in turn is linked to a number of "semantically similar" ones: dry is linked to parched, arid, dessicated and bone-dry and wet to soggy, waterlogged, etc. Semantically similar adjectives are "indirect antonyms" of the contral member of the opposite pole. Relational adjectives ("pertainyms") point to the nouns they are derived from (criminal-crime).

There are only few adverbs in WordNet (hardly, mostly, really, etc.) as the majority of English adverbs are straightforwardly derived from adjectives via morphological affixation (surprisingly, strangely, etc.)

1.5.1.3 Cross-POS relations

The majority of the WordNet's relations connect words from the same part of speech (POS). Thus, WordNet really consists of four sub-nets, one each for nouns, verbs, adjectives and adverbs, with few cross-POS pointers. Cross-POS relations include the “morphosemantic” links that hold among semantically similar words sharing a stem with the same meaning: observe (verb), observant (adjective) observation, observatory (nouns). In many of the noun-verb pairs the semantic role of the noun with respect to the verb has been specified: {sleeper, sleeping_car} is the LOCATION for {sleep} and {painter}is the AGENT of {paint}, while {painting, picture} is its RESULT.

More Information
Fellbaum, Christiane (2005). WordNet and wordnets. In: Brown, Keith et al. (eds.), Encyclopedia of Language and Linguistics, Second Edition, Oxford: Elsevier, 665-670.

1.5.2 VerbNet:

VerbNet (VN) (Kipper-Schuler 2006) is the largest on-line verb lexicon currently available for English. It is a hierarchical domain-independent, broad-coverage verb lexicon with mappings to other lexical resources such as WordNet (Miller, 1990; Fellbaum, 1998), Xtag (XTAG Research Group, 2001), and FrameNet (Baker et al., 1998). VerbNet is organized into verb classes extending Levin (1993) classes through refinement and addition of subclasses to achieve syntactic and semantic coherence among members of a class. Each verb class in VN is completely described by thematic roles, selectional restrictions on the arguments, and frames consisting of a syntactic description and semantic predicates with a temporal function, in a manner similar to the event decomposition of Moens and Steedman (1988). Each VN class contains a set of syntactic descriptions, or syntactic frames, depicting the possible surface realizations of the argument structure for constructions such as transitive, intransitive, prepositional phrases, resultatives, and a large set of diathesis alternations. Semantic restrictions (such as animate, human, organization) are used to constrain the types of thematic roles allowed by the arguments, and further restrictions may be imposed to indicate the syntactic nature of the constituent likely to be associated with the thematic role. Syntactic frames may also be constrained in terms of which prepositions are allowed. Each frame is associated with explicit semantic information, expressed as a conjunction of boolean semantic predicates such as `motion,' `contact,' or `cause.' Each semantic predicate is associated with an event variable E that allows predicates to specify when in the event the predicate is true (start(E) for preparatory stage, during(E) for the culmination stage, and end(E) for the consequent stage). Figure 1. shows a complete entry for a frame in VerbNet class Hit-18.1.

Figure 1: Simplified VerbNet entry for Hit-18.1 class

Class Hit-18.1			
Roles and Restrictions: Agent[+int_control] Patient[+concrete] Instrument[+concrete]			
Members: bang, bash, hit, kick, ...			
Frames:			
Name	Example	Syntax	Semantics
Basic Transitive	Paula hit the ball	Agent V Patient	cause(Agent, E)manner(during(E), directedmotion, Agent) ! contact(during(E), Agent, Patient) manner(end(E),forceful, Agent) contact(end(E), Agent, Patient)

VerbNet has recently been integrated with 57 new classes from Korhonen and Briscoe's (2004) (K&B) proposed extension to Levin's original classification (Kipper et al., 2006). This work has involved associating detailed syntactic-semantic descriptions to the K&B classes, as well as organizing them appropriately into the existing VN taxonomy. An additional set of 53 new classes from Korhonen and Ryant (2005) (K&R) have also been incorporated into VN. The outcome is a freely available resource which constitutes the most comprehensive and versatile Levin-style verb classification for English. After the two extensions VN has now also increased our coverage of PropBank tokens (Palmer et. al., 2005) from 78.45% to 90.86%, making feasible the creation of a substantial training corpus annotated with VN thematic role labels and class membership assignments, to be released in 2007. This will finally enable large-scale experimentation on the utility of syntax-based classes for improving the performance of syntactic parsers and semantic role labelers on new domains.

Integrating the two recent extensions to Levin classes into VerbNet was an important step in order to address a major limitation of Levin's verb classification, namely the fact that verbs taking ADJP, ADVP, predicative, control and sentential complements were not included or addressed in depth in that work. This limitation excludes many verbs that are highly frequent in language. A summary of how this integration affected VN and the result of the extended VN is shown in Table 1. The figures show that our work enriched and expanded VN considerably. The number of first-level classes grew significantly (from 191 to 274), there was also a significant increase in the number of verb senses and lemmas, along with the set of semantic predicates and the syntactic restrictions on sentential complements.

Table 1: Summary of the Lexicon's Extension

	Original VN	**Extended VN**
First-level classes	191	274
Thematic roles	21	23
Semantic predicates	64	94
Syntactic restrictions (on sentential compl)	3	55
Number of verb senses	4656	5257
Number of lemmas	3445	3769

Table 2: Thematic roles and example classes that use them

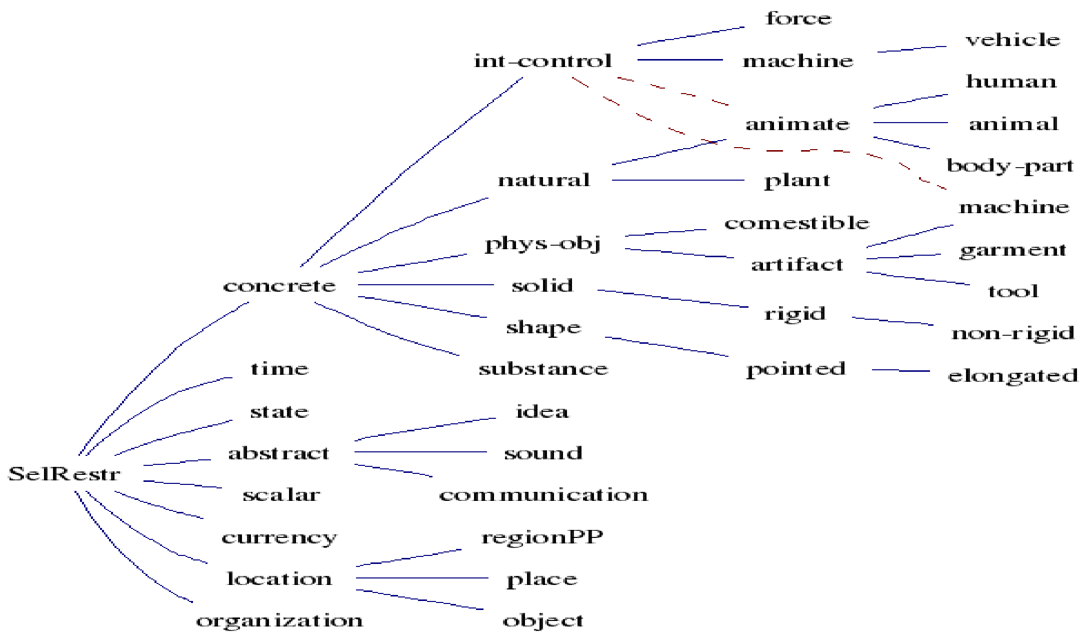
Actor:	used for some communication classes (e.g., Chitchat-37.6, Marry-36.2, Meet-36.2) when both arguments can be considered symmetrical (pseudo-agents).
Agent:	generally a human or an animate subject. Used mostly as a volitional agent, but also used in VerbNet for internally controlled subjects such as forces and machines.
Asset:	used for the Sum of Money Alternation, present in classes such as Build-26.1, Get-13.5.1, and Obtain-13.5.2 with `currency' as a selectional restriction.

Attribute:	attribute of Patient/Theme refers to a quality of something that is being changed, as in (The price)att of oil soared. At the moment, we have only one class using this role Calibratable cos-45.6 to capture the Possessor Subject Possessor-Attribute Factoring Alternation. The selectional restriction `scalar' (defined as a quantity, such as mass, length, time, or temperature, which is completely specified by a number on an appropriate scale) ensures the nature of Attribute.
Beneficiary:	the entity that benefits from some action. Used by such classes asBuild-26.1, Get-13.5.1, Performance-26.7, Preparing-26.3, and Steal-10.5. Generally introduced by the preposition `for', or double object variant in the benefactive alternation.
Cause:	used mostly by classes involving Psychological Verbs and Verbs Involving the Body.
Location, Destination, Source:	used for spatial locations.
Destination:	end point of the motion, or direction towards which the motion is directed. Used with a `to' prepositional phrase by classes of change of location, such as Banish-10.2, and Verbs of Sending and Carrying. Also used as location direct objects in classes where the concept of destination is implicit (and location could not be Source), such as Butter-9.9, or Image impression-25.1.
Source:	start point of the motion. Usually introduced by a source prepositional phrase (mostly headed by `from' or `out of'). It is also used as a direct object in such classes as Clear-10.3, Leave-51.2, and Wipe instr-10.4.2.
Location:	underspecified destination, source, or place, in general introduced by a locative or path prepositional phrase.
Experiencer:	used for a participant that is aware or experiencing something. In VerbNet it is used by classes involving Psychological Verbs, Verbs of Perception, Touch, and Verbs Involving the Body.
Extent:	used only in the Calibratable-45.6 class, to specify the range or degree of change, as in The price of oil soared (10%)ext. This role may be added to other classes.
Instrument:	used for objects (or forces) that come in contact with an object and cause some change in them. Generally introduced by a `with' prepositional phrase. Also used as a subject in the Instrument Subject Alternation and as a direct object in the Poke-19 class for the Through/With Alternation and in the Hit-18.1 class for the With/Against Alternation.
Material and Product:	used in the Build and Grow classes to capture the key semantic components of the arguments. Used by classes from Verbs of Creation and Transformation that allow for the Material/Product Alternation.
Material:	start point of transformation.
Product:	end result of transformation.
Patient:	used for participants that are undergoing a process or that have been affected in some way. Verbs that explicitly (or implicitly) express changes of state have Patient as their usual direct object. We also use Patient1 and Patient2 for some

	classes of Verbs of Combining and Attaching and Verbs of Separating and Disassembling, where there are two roles that undergo some change with no clear distinction between them.
Predicate:	used for classes with a predicative complement.
Recipient:	target of the transfer. Used by some classes of Verbs of Change of Possession, Verbs of Communication, and Verbs Involving the Body. The selection restrictions on this role always allow for animate and sometimes for organization recipients.
Stimulus:	used by Verbs of Perception for events or objects that elicit some response from an xperiencer. This role usually imposes no restrictions.
Theme:	used for participants in a location or undergoing a change of location. Also, Theme1 and Theme2 are used for a few classes where there seems to be no distinction between the arguments, such as Differ-23.4 and Exchange-13.6 classes.
Time:	class-specific role, used in Begin-55.1 class to express time.
Topic:	topic of communication verbs to handle theme/topic of the conversation or transfer of message. In some cases, like the verbs in the Say-37.7 class, it would seem better to have `Message' instead of `Topic', but we decided not to proliferate the number of roles.

Each verb argument is assigned one (usually unique) thematic role within the class. A few exceptions to this uniqueness are classes which contain verbs with symmetrical arguments, such as Chitchat-37.6 class, or the ContiguousLocation-47.8 class. These classes have indexed roles such as Actor1 and Actor2, as explained above.

Figure 2: Selectional Restrictions associated with thematic role



2. Computational Lexical Semantics

2.1 Word Similarity: Thesaurus Methods:

Synonymy is a binary relation between words; two words are either synonyms or not. For most computational purposes we use instead a looser metric of word similar WORD SIMILARITY or semantic distance. Two words are more similar if they share more features of SEMANTIC DISTANCE meaning, or are near-synonyms. Two words are less similar, or have greater semantic distance, if they have fewer common meaning elements. Although we have described them as relations between words, synonymy, similarity, and distance are actually relations between word senses. For example of the two senses of bank, we might say that the financial sense is similar to one of the senses of fund while the riparian sense is more similar to one of the senses of slope. In the next few sections of this chapter, we will need to compute these relations over both words and senses. The ability to compute word similarity is a useful part of many language understanding applications. In information retrieval or question answering we might want to retrieve documents whose words have similar meanings to the query words. In summarization, generation, and machine translation, we need to know whether two words are similar to know if we can substitute one for the other in particular contexts. In language modeling, we can use semantic similarity to cluster words for class-based models. One interesting class of applications for word similarity is automatic grading of student responses. For example algorithms for automatic essay grading use word similarity to determine if an essay is similar in meaning to a correct answer. We can also use word-similarity as part of an algorithm to take an exam, such as a multiple choice vocabulary test. Automatically taking exams is useful in test designs in order to see how easy or hard a particular multiple-choice question or exam is.

There are two classes of algorithms for measuring word similarity. This section focuses on **thesaurus-based** algorithms, in which we measure the distance between two senses in an on-line thesaurus like WordNet or MeSH. The next section focuses on **distributional** algorithms, in which we estimate word similarity by finding words that have similar distributions in a corpus.

The Thesaurus-based algorithms use the structure of the thesaurus to define word similarity. In principle we could measure similarity using any information available in a thesaurus (meronymy, glosses, etc). In practice, however, thesaurus-based word similarity algorithms generally use only the hypernym/hyponym(*is-a* or subsumption) hierarchy. In WordNet, verbs and nouns are in separate hypernym hierarchies, so a thesaurus-based algorithm for WordNet can thus only compute noun-

noun similarity, or verb-verb similarity; we can't compare nouns to verbs, or do anything with adjectives or other parts of speech.
Resnik (1995) and Budanitsky and Hirst (2001) draw the important distinction between **word similarity** and **word relatedness**. Two words are similar if they are near-synonyms, or roughly substitutable in context. Word relatedness characterizes a larger set of potential relationships between words; antonyms, for example, have high relatedness,
but low similarity. The words *car* and *gasoline* are very related, but not similar, while *car* and *bicycle* are similar. Word similarity is thus a subcase of word relatedness. In general, the five algorithms we describe in this section do not attempt to distinguish between similarity and semantic relatedness; for convenience we will call them *similarity* measures, although some would be more appropriately described as relatedness measures.

Section 20.6. Word Similarity: Thesaurus Methods 17
The oldest and simplest thesaurus-based algorithms are based on the intuition that the shorter the **path** between two words or senses in the graph defined by the thesaurus hierarchy, the more similar they are. Thus a word/sense is very similar to its parents or its siblings, and less similar to words that are far away in the network. This notion can
be operationalized by measuring the number of edges between the two concept nodes in the thesaurus graph. Fig. 20.6 shows an intuition; the concept *dime* is most similar to *nickel* and *coin*, less similar to *money*, and even less similar to *Richter scale*. Formally, we specify path length as follows:
pathlen(*c*1,*c*2) = the number of edges in the shortest path in the thesaurus graph between the sense nodes *c*1 and *c*2

Path-based similarity can be can be defined just as the path length, often with a log transform (Leacock and Chodorow, 1998), resulting in the following common definition of **path length based similarity**:
$\text{sim}_{\text{path}}(c1,c2) = -\log \text{pathlen}(c1,c2)$
For most applications, we don't have sense-tagged data, and thus we need our algorithm to give us the similarity between words rather than between senses or concepts.
For any of the thesaurus-based algorithms, following Resnik (1995), we can approximate the correct similarity (which would require sense disambiguation) by just using the pair of senses for the two words that results in maximum sense similarity. Thus based on sense similarity we can define **word similarity** as follows:

$$\textbf{wordsim}(w1,w2) = \max_{\substack{c1\in \text{senses}(w1)\\ c2\in\text{senses}(w2)}} \text{sim}(c1,c2)$$

The basic path-length algorithm makes the implicit assumption that each link in the network represents a uniform distance. In practice, this assumption is not appropriate. Some links (for example those that are very deep in the WordNet hierarchy) often seem to represent an intuitively narrow distance, while other links (e.g., higher up in the WordNet hierarchy) represent an intuitively wider distance. For example, in Fig. 20.6, the distance from *nickel* to *money* (5) seems intuitively much shorter than the distance from *nickel* to an abstract word *standard*; the link between *medium of exchange* and *standard* seems wider than that between, say, *coin* and *coinage*. It is possible to refine path-based algorithms with normalizations based on depth in the hierarchy (Wu and Palmer, 1994), but in general we'd like an approach which lets us represent the distance associated with each edge independently.
A second class of thesaurus-based similarity algorithms attempts to offer just such a fine-grained metric. These **information content word similarity** algorithms still rely on the structure of the thesaurus, but also add probabilistic information derived from a corpus.

Using similar notions to those we introduced earlier to define soft selectional restrictions, let's first define P(c), following Resnik (1995), as the probability that a randomly selected word in a corpus is an instance of concept c (i.e., a separate random variable, ranging over words, associated with each concept). This implies that P(root) = 1, since any word is subsumed by the root concept. Intuitively, the lower a concept in the hierarchy, the lower its probability. We train these probabilities by counting in a corpus; each word in the corpus counts as an occurrence of each concept that contains it. For example, in Fig. 20.6 above, an occurrence of the word dime would count toward the frequency of coin, currency, standard, etc. More formally,
Resnik computes P(c) as follows:

$$P(c) = \frac{\sum_{w \in \text{words}(c)} count(w)}{N}$$

where word(c) is the set of words subsumed by concept *c*, and *N* is the total number of words in the corpus that are also present in the thesaurus. Fig. 20.7, from Lin (1998b), shows a fragment of the WordNet concept hierarchy
augmented with the probabilities *P*(*c*).
We now need two additional definitions. First, following basic information theory,
we define the information content (IC) of a concept *c* as:

$$\text{IC}(c) = -\log P(c)$$

Second, we define the **lowest common subsumer** or **LCS** of two concepts:

LCS(*c*1,*c*2) = the lowest common subsumer, i.e., the lowest node in the hierarchy that subsumes (is a hypernym of) both *c*1 and *c*2
There are now a number of ways to use the information content of a node in a word similarity metric. The simplest way was first proposed by Resnik (1995). We think of the similarity between two words as related to their common information; the more two words have in common, the more similar they are. Resnik proposes to estimate the
common amount of information by the **information content of the lowest common subsumer of the two nodes**. More formally, the **Resnik similarity** measure is:

Lin extended the Resnik intuition by pointing out that a similarity metric between objects A and B needs to do more than measure the amount of information in common between A and B. For example, he pointed out that in addition, the more **differences** between A and B, the less similar they are.

Lin measures the commonality between A and B as the information content of the proposition that states the commonality between A and B:

IC(Common(A,B))

He measures the difference between A and B as

IC(description(A,B))−IC(common(A,B))

where description(A,B) describes A and B. Given a few additional assumptions about similarity, Lin proves the following theorem:
Similarity Theorem: The similarity between A and B is measured by the ratio between the amount of information needed to state the commonality of A and B and the information needed to fully describe what A and B are:

$$\mathrm{sim}_{\mathrm{Lin}}(A,B) = \frac{\log P(\text{common(A,B)})}{\log P(\text{description(A,B)})}$$

Applying this idea to the thesaurus domain, Lin shows (in a slight modification of Resnik's assumption) that the information in common between two concepts is twice the information in the lowest common subsumer LCS($c1$,$c2$). Adding in the above definitions of the information content of thesaurus concepts,the final **Lin similarity**
function is:

$$\mathrm{sim}_{\mathrm{Lin}}(c_1,c_2) = \frac{2\times\log P(LCS(c_1,c_2))}{\log P(c_1)+\log P(c_2)}$$

For example, using $\mathrm{sim}_{\mathrm{lin}}$, Lin (1998b) shows that the similarity between the concepts of *hill* and *coast* from Fig. 20.7 is

$$\mathrm{sim}_{\mathrm{Lin}}(\text{hill},\text{coast}) = \frac{2\times\log P(\text{geological-formation})}{\log P(\text{hill})+\log P(\text{coast})} = 0.59$$

A very similar formula, **Jiang-Conrath distance** (Jiang and Conrath, 1997) (al-though derived in a completely different way from Lin, and expressed as a distance rather than similarity function) has been shown to work as well or better than all the other thesaurus-based methods

$$\mathrm{dist}_{\mathrm{JC}}(c_1,c_2) = 2\times\log P(\mathrm{LCS}(c_1,c_2)) - (\log P(c_1)+\log P(c_2))$$

$\mathrm{dist}_{\mathrm{JC}}$ can be transformed into a similarity by taking the reciprocals.

Finally, we describe a dictionary-based method, an extension of the Lesk algorithm for word-sense disambiguation described in Sec. 20.4.1. We call this a dictionary rather than a thesaurus method because it makes use of glosses, which are in general a property of dictionaries rather than thesauri (although WordNet does have glosses). Like the Lesk algorithm, the intuition of this **Extended Gloss Overlap**, or **Extended Lesk** measure (Banerjee and Pedersen, 2003) is that two concepts/senses in a thesaurus are similar if their glosses contain overlapping words. We'll begin by sketching an overlap function for two glosses. Consider these two concepts, with their glosses:

- drawing paper: <u>paper</u> that is <u>specially prepared</u> for use in drafting
- *decal:* the art of transferring designs from <u>specially prepared paper</u> to a wood or glass or metal surface.

For each n-word phrase that occurs in both glosses, Extended Lesk adds in a score of *n*2 (the relation is non-linear because of the Zipfian relationship between lengths of phrases and their corpus frequencies; longer overlaps are rare so should be weighted more heavily). Here the overlapping phrases are *paper* and *specially prepared*, for a total similarity score of $1^2+2^2 = 5$.
Given such an overlap function, when comparing two concepts (synsets), Extended Lesk not only looks for overlap between their glosses, but also between the glosses of the senses which are hypernyms, hyponyms, meronyms, and other relations of the two concepts. For example if we just considered hyponyms, and defined gloss(hypo(A)) as
the concatenation of all the glosses of all the hyponymsenses of A, the total relatedness between two concepts A and B might be:
similarity(A,B) = overlap(gloss(A), gloss(B)) + overlap(gloss(hypo(A)), gloss(hypo(B)))+ overlap(gloss(A), gloss(hypo(B))) + overlap(gloss(hypo(A)),gloss(B))
Let RELS be the set of possible WordNet relations whose glosses we compare; assuming a basic overlap measure as sketched above, we can then define the **Extended Lesk** overlap measure as:

$$\mathrm{sim}_{\mathrm{eLesk}}(c_1,c_2) = \sum_{r,q\in\mathrm{RELS}} \mathrm{overlap}(\mathrm{gloss}(r(c_1)),\mathrm{gloss}(q(c_2)))$$

$$
\begin{aligned}
\text{sim}_{\text{path}}(c_1,c_2) &= -\log \text{pathlen}(c_1,c_2)\\
\text{sim}_{\text{Resnik}}(c_1,c_2) &= -\log P(\text{LCS}(c_1,c_2))\\
\text{sim}_{\text{Lin}}(c_1,c_2) &= \frac{2\times \log P(LCS(c_1,c_2))}{\log P(c_1)+\log P(c_2)}\\
\text{sim}_{\text{jc}}(c_1,c_2) &= \frac{1}{2\times \log P(\text{LCS}(c_1,c_2)) - (\log P(c_1)+\log P(c_2))}\\
\text{sim}_{\text{eLesk}}(c_1,c_2) &= \sum_{r,q\in \text{RELS}} \text{overlap}(\text{gloss}(r(c_1)),\text{gloss}(q(c_2)))
\end{aligned}
$$

Figure 20.8 Five thesaurus-based (and dictionary-based) similarity measures.

Evaluating Thesaurus-based Similarity:
Which of these similarity measures is best?
Word similarity measures have been evaluated in two ways. One intrinsic method is to compute the correlation coefficient between word similarity scores from an algorithm and word similarity ratings assigned by humans; such human ratings have been obtained for 65 word pairs by Rubenstein and Goodenough (1965), and 30 word pairs by Miller and Charles (1991). Another more extrinsic evaluation method is to embed the similarity measure in some end application like detection of **malapropisms** (real-word spelling errors) (Budanitsky and Hirst, 2006; Hirst and Budanitsky, 2005), or other NLP applications like word-sense disambiguation (Patwardhan et al., 2003; McCarthy et al., 2004) and evaluate its impact on end-to-end performance. All of these evaluations suggest that all the above measures perform relatively well, and that of these, Jiang-Conrath similarity and Extended Lesk similarity are two of the best approaches,
depending on the application.

Word Similarity: Distributional Methods
The previous section showed how to compute similarity between any two senses in a thesaurus, and by extension between any two words in the thesaurus hierarchy. But of course we don't have such thesauri for every language. Even for languages where we do have such resources, thesaurus-based methods have a number of limitations. The
obvious limitation is that thesauri often lack words, especially new or domain-specific words. In addition, thesaurus-based methods only work if rich hyponymy knowledge is present in the thesaurus. While we have this for nouns, hyponym information for verbs tends to be much sparser, and doesn't exist at all for adjectives and adverbs.
Finally, it is more difficult with thesaurus-based methods to compare words in different hierarchies, such as nouns with verbs. For these reasons, methods which can automatically extract synonyms and other word relations from corpora have been developed. In this section we introduce such **distributional** methods, which can be applied directly to supply a word relatedness measure for NLP tasks. Distributional methods can also be used for **automatic thesaurus generation** for automatically populating or augmenting on-line thesauruses like WordNet with new synonyms and, as we will see in Sec. 20.8, with other relations like hyponymy and meronymy.
The intuition of distributional methods is that the meaning of a word is related to the distribution of words around it; in the famous dictum of Firth (1957), "You shall know a word by the company it keeps!". Consider the following example, modified by Lin (1998a) from (?):

A bottle of *tezguino* is on the table.

Everybody likes *tezguino*.
Tezguino makes you drunk.
We make *tezguino* out of corn.

The context in which tezguino occurs suggest that it might be some kind of fermented alcoholic drink made of corn.The distributional method tries to capture this intuition by representing features of the context of *tezguino* that might overlap with features of similar words like *beer*, *liquor*, *tequila*, and so on. For example such features might be occurs before *drunk* or occurs after *bottle* or is the direct object of *likes*.

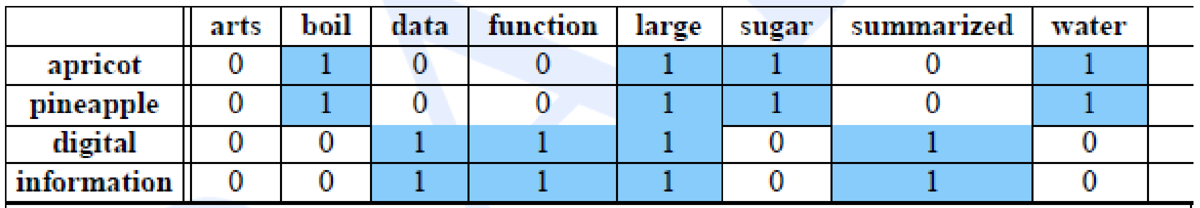
We can then represent a word *w* as a **feature vector.** For example, suppose we had one binary feature *fi* representing each of the *N* words in the lexicon *vi*. The feature means *w* occurs in the neighborhood of word *vi*, and hence takes the value 1 if *w* and *vi* occur in some context window, and 0 otherwise. We could represent the meaning of word *w* as the feature vector.

$$\vec{w} = (f_1, f_2, f_3, \cdots, f_N)$$

If *w*= *tezguino*, *v*1=*bottle*, *v*2=*drunk*, and *v*3=*matrix*, the co-occurrence vector for *w* from the corpus above would be:

$$\vec{w} = (1, 1, 0, \cdots)$$

Given two words represented by such sparse feature vectors, we can apply a vector distance measure and say that the words are similar if the two vectors are close by this measure. Fig. 20.9 shows an intuition about vector similarity for the four words *apricot*, *pineapple*, *digital*, and *information*. Based on the meanings of these four
words, we would like a metric that shows *apricot* and *pineapple* to be similar, *digital* and *information*, to be similar, and the other four pairings to produce low similarity. For each word, Fig. 20.9 shows a short piece (8 dimensions) of the (binary) word co occurrence vectors, computed from words that occur within a two-line context in the Brown corpus. The reader should convince themselves that the vectors for *apricot* and *pineapple* are indeed more similar than those of, say, *apricot* and *information*. For pedagogical purposes we've shown the context words that are particularly good at discrimination. Note that since vocabularies are quite large (10,000-100,000 words) and most words don't occur near each other in any corpus, real vectors are quite sparse.



	arts	boil	data	function	large	sugar	summarized	water
apricot	0	1	0	0	1	1	0	1
pineapple	0	1	0	0	1	1	0	1
digital	0	0	1	1	1	0	1	0
information	0	0	1	1	1	0	1	0

Figure 20.9 Co-occurrence vectors for four words, computed from the Brown corpus, showing only 8 of the (binary) dimensions (hand-picked for pedagogical purposes to show discrimination). Note that *large* occurs in all the contexts and *arts* occurs in none; a real vector would be extremely sparse.

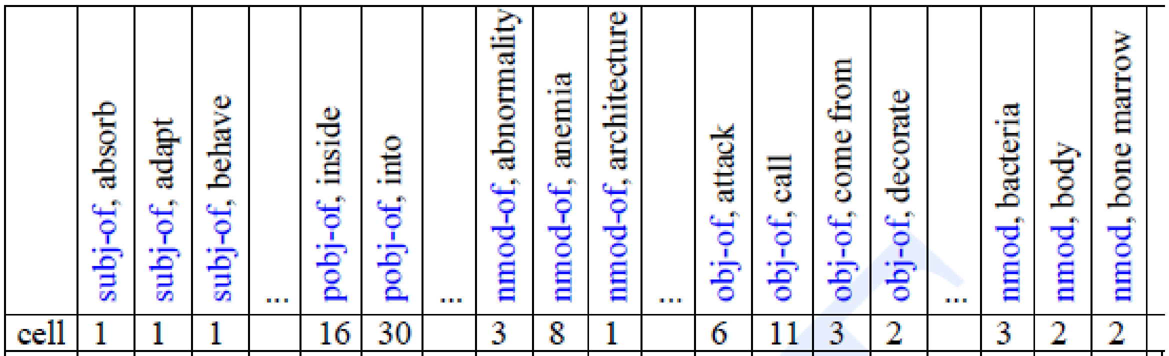
let's move on to examine the details of these measures. Specifying a distributional similarity measure requires that we specify three parameters: (1) how the co-occurrence terms are defined (i.e. what counts as a neighbor), (2) how these terms are weighted (binary? frequency? mutual information?) and (3) what vector distance metric we use (cosine? Euclidean distance?). Let's look at each of these requirements in the next three subsections.

Defining a Word's Co-occurrence Vectors

In our example feature vector, we used the feature *w* occurs in the neighborhood of word *vj*. That is, for a vocabulary size *N*, each word *w* had *N* features, specifying whether vocabulary element *vj* occurred in the neighborhood. Neighborhoods range from a small window of words (as few as one or two words on either side) to very large windows of ±500 words. In a minimal window, for example, we might have two features for each word *vj* in the vocabulary, word *vk* occurs immediately before word *w* and word *vk* occurs immediately after word *w*. To keep these contexts efficient, we often ignore very frequent words which tend not to be very discriminative, e.g., function words such as *a*, *am, the, of, 1, 2,* and so on. These removed words are called **stopwords** or the **stoplist**. Even with the removal of the stopwords, when used on very large corpora these co occurrence vectors tend to be very large. Instead of using every word in the neighborhood, Hindle (1990) suggested choosing words that occur in some sort of **grammatical relation** or **dependency** to the target words. Hindle suggested that nouns which bear the same grammatical relation to the same verb might be similar. For example, the words *tea*, *water*, and *beer* are all frequent direct objects of the verb *drink*. The words *senate*, *congress, panel*, and *legislature* all tend to be subjects of the verbs *consider*, *vote*, and *approve*.
There have been a wide variety of realizations of Hindle's idea since then. In general, in these methods each sentence in a large corpus is parsed and a dependency parse is extracted. We saw in Ch. 12 lists of grammatical relations produced by dependency parsers, including noun-verb relations like subject, object, indirect object, and noun-noun relations like genitive, ncomp, and so on. A sentence like the following would result in the set of dependencies shown here:
I discovered dried tangerines
discover (subject I) I (subj-of discover)
tangerine (obj-of discover) tangerine (adj-mod dried)
dried(adj-mod-of-tangerine)
Since each word can be in a varity of different dependency relations with other words, we'll need to augment the feature space. Each feature is now a pairing of a word and a relation, so instead of a vector of *N* features, we have a vector of *N* ×*R* features, where *R* is the number of possible relations. Fig. 20.10 shows a schematic example of such a vector, taken from Lin (1998a), for the word *cell*. As the value of each attribute we have shown the frequency of the feature co-occurring with *cell*; the next section will discuss the use of what values and weights to use for each attribute. Since full parsing is very expensive, it is common to use a chunker or shallow parser.
With the goal of extracting only a smaller set of relations like subject, direct object, and prepositional object of a particular preposition (Curran,2003).

Measures of Association with Context

We have a definition for the features or dimensions of a word's context vector, we are ready to discuss the values that should be associated with those features. These values are typically thought of as **weights** or measures of **association** between each target word *w* and a given feature *f* . In the example in Fig. 20.9, our association
measure was a binary value for each feature, 1 if the relevant word had occurred in the context, 0 if not. In the example in Fig. 20.10, we used a richer association measure, the relative frequency with which the particular context feature had co-occurred with the target word.



	subj-of, absorb	subj-of, adapt	subj-of, behave	...	pobj-of, inside	pobj-of, into	...	nmod-of, abnormality	nmod-of, anemia	nmod-of, architecture	...	obj-of, attack	obj-of, call	obj-of, come from	obj-of, decorate	...	nmod, bacteria	nmod, body	nmod, bone marrow
cell	1	1	1		16	30		3	8	1		6	11	3	2		3	2	2

Figure 20.10 Co-occurrence vector for the word *cell*, from Lin (1998a), showing grammatical function (dependency) features. Values for each attribute are frequency counts from a 64-million word corpus, parsed by an early version of MINIPAR.

Frequency, or probability, is certainly a better measure of association than just a binary value; features that occur often with a target word are more likely to be good indicators of the word's meaning. Let's define some terminology for implementing a probabilistic measure of association. For a target word *w*, each element of its co occurrence
vector is a feature *f* , consisting of a relation *r* and a related word *w*′; we can say *f* = (*r*,*w*′). For example, one of the features of the word *cell* in Fig. 20.10 is *f* = (*r*,*w*′) =(obj-of, *attack*).

The probability of a feature *f* given a target word *w* is *P*(*f* |*w*), for which the maximum likelihood estimate is

$$P(f|w) = \frac{\text{count}(f,w)}{\text{count}(w)}$$

Similarly the maximum likelihood estimate for the joint probability P(f ,w) is:

$$P(f,w) \quad = \quad \frac{\text{count}(f,w)}{\sum_{w'} \text{count}(f,w'))}$$

P(w) and P(f) are computed similarly.
Thus if we were to define simple probability as a measure of association it would look as follows:

$$\text{assoc}_{\text{prob}}(w,f) = P(f|w)$$

It turns out, however, that simple probability doesn't work as well as more sophisticated association schemes for word similarity. We, therefore, need a weighting or measure of association which asks how much more often than chance that the feature co-occurs with the target word. As Curran (2003) points out, such a weighting is what we also want for finding good **collocations**, and so the measures of association used for weighting context words for semantic similarity are exactly the same measure used for finding a word's collocations. One of the most important measures of association was first proposed by Church and Hanks (1989, 1990) and is based on the notion of **mutual information**. The **mutual information** between two random variables *X* and *Y* is

$$I(X,Y) = \sum_x \sum_y P(x,y) \log_2 \frac{P(x,y)}{P(x)P(y)}$$

The **pointwise mutual information** (Fano, 1961)[3] is a measure of how often two events *x* and *y* occur, compared with what we would expect if they were independent

$$I(x,y) = \log_2 \frac{P(x,y)}{P(x)P(y)}$$

We can apply this intuition to co-occurrence vectors, by defining the pointwise mutual information association between a target word *w* and a feature *f* as

$$\text{assoc}_{\text{PMI}}(w,f) = \log_2 \frac{P(w,f)}{P(w)P(f)}$$

The intuition of the PMI measure is that the numerator tells us how often we observed the two words together (assuming we compute probability using MLE as above). The denominator tells us how often we would **expect** the two words to co-occur assuming they each occurred independently, so their probabilities could just be multiplied.

For both assoc$_{\text{PMI}}$ and assoc$_{\text{Lin}}$, we generally only use the feature *f* for a word *w* if the assoc value is positive, since negative PMI values (implying things are co occurring less often than we would expect by chance) tend to be unreliable unless the training corpora are enormous (Dagan et al., 1993; Lin, 1998a). In addition, when we are using the assoc-weighted features to compare two target words, we only use features that co-occur with both target words.
Fig 20.11 from Hindle (1990) shows the difference between raw frequency counts and PMI-style association, for some direct objects of the verb *drink*.

Object	Count	PMI assoc	Object	Count	PMI assoc
bunch beer	2	12.34	wine	2	9.34
tea	2	11.75	water	7	7.65
Pepsi	2	11.75	anything	3	5.15
champagne	4	11.75	much	3	5.15
liquid	2	10.53	it	3	1.25
beer	5	10.20	<SOME AMOUNT>	2	1.22

Figure 20.11 Objects of the verb *drink*, sorted by PMI, from Hindle (1990).

One of the most successful association measures for word similarity attempts to capture the same intuition as mutual information, but uses the **t-test** statistic to measure how much more frequent the association is than chance. This measure was proposed for collocation-detection by Manning and Schutze (1999, Chapter 5) and then applied
to word similarity by Curran and Moens (2002), Curran (2003). The t-test statistic computes the difference between observed and expected means, normalized by the variance. The higher the value of *t*, the more likely we can reject the null hypothesis that the observed and expected means are the same.

$$t = \frac{\bar{x} - \mu}{\sqrt{\frac{s^2}{N}}}$$

When applied to association between words, the null hypothesis is that the two words are independent, and hence $P(f,w) = P(f)P(w)$ correctly models the relationship between the two words. We want to know how different the actual MLE probability $P(f,w)$ is from this null hypothesis value, normalized by the variance. Note the similarity to the comparison with the product model in the PMI measure above. The variance *s*2 can be approximated by the expected probability

P(*f*)*P*(*w*) (see Manning and Schutze (1999)). Ignoring *N* (since it is constant), the resulting t-test association measure from Curran(2003) is thus

$$\text{assoc}_{\text{t-test}}(w,f) = \frac{P(w,f) - P(w)P(f)}{\sqrt{P(f)P(w)}}$$

Defining Similarity between two vectors
To define similarity between two target words *v* and *w*, we need a measure for taking two such vectors and giving a measure of vector similarity. Perhaps the simplest two measures of vector distance are the Manhattan and Euclidean distance. Fig. 20.12 shows a graphical intuition for Euclidean and Manhattan distance between two two dimensional vectors ~*a* and ~*b*. The **Manhattan distance**, also known as **Levenshtein distance** or **L1 norm**, is

$$\text{distance}_{\text{manhattan}}(\vec{x},\vec{y}) = \sum_{i=1}^{N} |x_i - y_i|$$

The **Euclidean distance**, also called the **L2 norm** is:

$$\text{distance}_{\text{euclidean}}(\vec{x},\vec{y}) = \sqrt{\sum_{i=1}^{N} (x_i - y_i)^2}$$

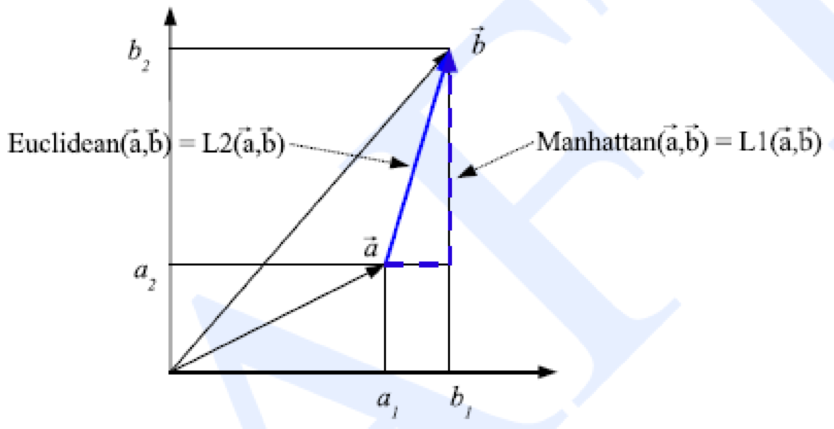

b_2
$\vec{b}$
$\text{Euclidean}(\vec{a},\vec{b}) = \text{L2}(\vec{a},\vec{b})$
$\text{Manhattan}(\vec{a},\vec{b}) = \text{L1}(\vec{a},\vec{b})$
$\vec{a}$
a_2
a_1
b_1

Figure 20.12 The Euclidean and Manhattan distance metrics for vectors $a = (a_1, a_2)$, and $b = (b_1, b_2)$, just to give the reader a grpahical intuition about the idea of distance between vectors; these particular metrics are generally not used for word similarity. See Ch. 9 for more on distance metrics.

Although the Euclidean and Manhattan distance metrics provide a nice geometric intuition for vector similarity and distance, these measures are rarely used for word similarity. If we assume a feature vector is a **binary vector**, we can define such a similarity metric as follows, using the **dot product** or **inner product** operator from

$$\text{sim}_{\text{dot-product}}(\vec{v},\vec{w}) = \vec{v}\cdot\vec{w} = \sum_{i=1}^{N} v_i \times w_i$$

In most cases, though, as we saw in the previous section, the values of our vector are not binary. Let’s assume for the rest of this section that the entries in the co-occurrence vector are the

association values between the target words and each of the features. In other words, let's define the vector for a target word ~*w* with *N* features *f*1.. *fN* as:

$$\vec{w} = (\text{assoc}(w, f_1), \text{assoc}(w, f_2), \text{assoc}(w, f_3), \ldots, \text{assoc}(w, f_N))$$

Now we can apply $\text{sim}_{\text{dot-product}}$ to vectors with values defined as associations, to get the dot-product similarity between weighted values. This raw dot-product, however, has a problem as a similarity metric: it favors **long** vectors. The **vector length** is defined as:

$$|\vec{v}| = \sqrt{\sum_{i=1}^{N} v_i^2}$$

A vector can be longer because it has more non-zero values, or because each dimension has a higher value. Both of these facts will increase the dot product. It turns out that both of these can occur as a by-product of word frequency. A vector from a very frequent word will have more non-zero co-occurrence association values, and will probably have higher values in each (even using association weights that control somewhat for frequency). The raw dot product thus favors frequent words.

We need to modify the dot product to normalize for the vector length. The simplest way is just to divide the dot product by the lengths of each of the two vectors. This **normalized dot product** turns out to be the same as the cosine of the angle between the two vectors. The **cosine** or normalized dot product similarity metric is thus:

$$\text{sim}_{\text{cosine}}(\vec{v}, \vec{w}) = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i \times w_i}{\sqrt{\sum_{i=1}^{N} v_i^2} \sqrt{\sum_{i=1}^{N} w_i^2}}$$

$$\text{assoc}_{\text{prob}}(w, f) = P(f|w) \qquad (20.35)$$

$$\text{assoc}_{\text{PMI}}(w, f) = \log_2 \frac{P(w,f)}{P(w)P(f)} \qquad (20.38)$$

$$\text{assoc}_{\text{Lin}}(w, f) = \log_2 \frac{P(w,f)}{P(w)P(r|w)P(w'|w)} \qquad (20.39)$$

$$\text{assoc}_{\text{t-test}}(w, f) = \frac{P(w,f) - P(w)P(f)}{\sqrt{P(f)P(w)}} \qquad (20.41)$$

$$\text{sim}_{\text{cosine}}(\vec{v}, \vec{w}) = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i \times w_i}{\sqrt{\sum_{i=1}^{N} v_i^2} \sqrt{\sum_{i=1}^{N} w_i^2}} \qquad (20.47)$$

$$\text{sim}_{\text{Jaccard}}(\vec{v}, \vec{w}) = \frac{\sum_{i=1}^{N} \min(v_i, w_i)}{\sum_{i=1}^{N} \max(v_i, w_i)} \qquad (20.48)$$

$$\text{sim}_{\text{Dice}}(\vec{v}, \vec{w}) = \frac{2 \times \sum_{i=1}^{N} \min(v_i, w_i)}{\sum_{i=1}^{N} (v_i + w_i)} \qquad (20.49)$$

$$\text{sim}_{\text{JS}}(\vec{v}||\vec{w}) = D(\vec{v}|\tfrac{\vec{v}+\vec{w}}{2}) + D(\vec{w}|\tfrac{\vec{v}+\vec{w}}{2}) \qquad (20.52)$$

Figure 20.13 Defining word similarity: measures of association between a target word w and a feature $f = (r, w')$ to another word w', and measures of vector similarity between word co-occurrence vectors $\vec{v}$ and $\vec{w}$.

Evaluating Distributional word Similarity

Distributional similarity can evaluated in the same ways as thesaurus-based similarity; we can compare intrinsically to human similarity scores, or we can evaluate it extrinsically as part of end-to-end applications. Besides word sense disambiguation and malapropism detection, similarity measures have been used as a part of systems for the grading of exams and essays(Landauer et al., 1997), or taking TOEFL multiple choice exams (Landauer and Dumais, 1997; Turney et al., 2003). Let *S* be the set of words that are defined as similar in the thesaurus, by being in the same synset, or perhaps sharing the same hypernym, or being in the hypernym-hyponym relation. Let *S′* be the
set of words that are classified as similar by some algorithm. We can define precision and recall as:

$$\textbf{precision} = \frac{|S \cap S'|}{|S'|} \textbf{recall} = \frac{|S \cap S'|}{|S|}$$

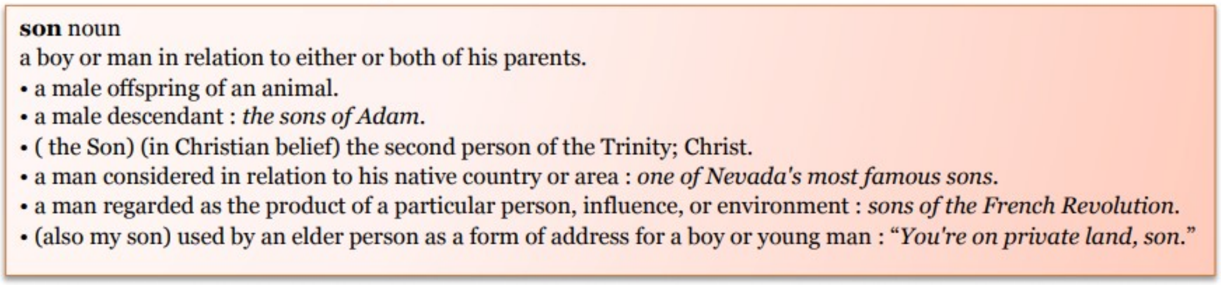
Curran (2003) evaluated a number of distributional algorithms using comparison with thesauri and found that the Dice and Jaccard methods performed best as measures of vector similarity, while t-test performed best as a measure of association. Thus the best metric weighted the associations with t-test, and then used either Dice or Jaccard
to measure vector similarity.

Lexemes:

A lexicon generally has a highly structured form. It stores the meanings and uses of each word. It encodes the relations between words and meanings.

A lexeme is the minimal unit represented in the lexicon. It pairs a stem (the orthographic/phonological form chosen to represent words) with a symbolic form for meaning representation (sense).

A dictionary is a kind of lexicon where meanings are expressed through definitions and examples.



son noun
a boy or man in relation to either or both of his parents.
- a male offspring of an animal.
- a male descendant : *the sons of Adam.*
- (the Son) (in Christian belief) the second person of the Trinity; Christ.
- a man considered in relation to his native country or area : *one of Nevada's most famous sons.*
- a man regarded as the product of a particular person, influence, or environment : *sons of the French Revolution.*
- (also my son) used by an elder person as a form of address for a boy or young man : *"You're on private land, son."*

Several kinds of relationships can be defined between lexemes and senses (some of them are important for automatic processing).

Homonymy

It is a relation between words that have the same form (and the same POS) but unrelated meanings e.g. bank (the financial institution, the river bank).

It causes ambiguities for the interpretation of a sentence since it defines a set of different lexemes with the same orthographic form (bank1 , bank2,..). Related properties are homophony (same pronunciation but different orthography, e.g. be-bee) and homography (same orthography but different pronunciation pésca/pèsca)

Polysemy

It happens when a lexeme has more related meanings .It depends on the word etymology (unrelated meanings usually have a different origin) - e.g. bank/data bank/blood bank.

For polysemous lexemes we need to manage all the meanings. We should define a method to determine the meanings (their number and semantics) and if they are really distinct (by experts in lexicography) .We need to describe the eventual correlations among the meanings. We need to define how the meanings can be distinguished in order to attach the correct meaning to a word in a given context (word sense disambiguation).

Synonymy

It is a relationship between two distinct lexemes with the same meaning (i.e. they can be substituted for one another in a given context without changing its meaning and correctness) – e.g. I received a gift/present

The substitutability may not be valid for any context due to small semantic differences (e.g. price/fare of a service – the bus fare/the ticket price) . In general substitutability depends on the "semantic intersection" of the senses of the two lexemes and, in same cases, also by social factors (father/dad).

Hyponymy is a relationship between two lexemes (more precisely two senses) such that one denotes a subclass of the other

▫ car, vehicle – shark, fish – apple, fruit

▫ The relationship is not symmetric

 The more specialized concept is the hyponym of the more general one

 The more general concept is the hypernym of the more specialized one

Hyponymy (hypernymy) is the basis for the definition of a taxonomy (a tree structure that defines inclusion relationships in an object ontology) even if it is not properly a taxonomy

 The definition of a formal taxonomy would require a more uniform/rigorous
formalism in the interpretation of the inclusion relationship

 However the relationship defines a inheritance mechanism of the properties
from the ancestors of a given a concept in the hierarchy

Metaphor

 Situations where we refer to, and reason about, concepts using words and phrases whose meanings are
appropriate to other completely different kinds of concepts.

– Love is a rose. Time is money.

 Conventional metaphors

– That doesn't scare Digital, which has grown to be the world's second-largest computer maker by poaching customers of IBM's mid-range machines.

– COMPANY AS PERSON metaphor

– Fuqua Industries Inc. said Triton Group Ltd., a company it helped resuscitate, has begun acquiring Fuqua shares.

– And Ford was hemorrhaging; its losses would hit $1.54 billion in 1980.

Metonymy

 Situations where we denote a concept by naming some other concept closely related to it.

– He likes Shakespeare.

AUTHOR FOR AUTHOR'S WORKS

– The White House had no comment.

PLACE FOR INSTITUTION

Word-Sense Disambiguation

Word sense ambiguity is a central problem for many established Human Language Technology applications (e.g., machine translation, information extraction, question answering, information retrieval, text classification, and text summarization) Ide1998. This is also the case for associated subtasks (e.g., reference resolution, acquisition of sub categorization patterns, parsing, and,

obviously, semantic interpretation). For this reason, many international research groups are working on WSD, using a wide range of approaches. However, to date, no large-scale, broad-coverage, accurate WSD system has been built Snyder2004. With current state-of-the-art accuracy in the range 60-70%, WSD is one of the most important open problems in NLP.
Word Sense Disambiguation (WSD), has been a trending area of research in Natural Language Processing and Machine Learning. WSD is basically solution to the ambiguity which arises due to different meaning of words in different context.

For example, consider the two sentences.

“The bank will not be accepting cash on Saturdays. ”

“The river overflowed the bank.”
The word bank in the first sentence refers to the commercial (finance) banks, while in second sentence, it refers to the river bank. The ambiguity that arises due to this, is tough for a machine to detect and resolve. Detection of ambiguity is the first issue and resolving it and displaying the correct output is the second issue. Here, the code presented is for solving the second issue. Feel free to contribute to it.

```
import nltk
import codecs
from nltk.tokenize import PunktSentenceTokenizer,sent_tokenize, word_tokenize
from nltk.corpus import stopwords, wordnet
from nltk.stem import WordNetLemmatizer, PorterStemmer

def simpleFilter(sentence):

        filtered_sent = []
        lemmatizer = WordNetLemmatizer()
        stop_words = set(stopwords.words("english"))
        words = word_tokenize(sentence)

        for w in words:
        if w not in stop_words:
                filtered_sent.append(lemmatizer.lemmatize(w))

  return filtered_sent
```

The simpleFilter function takes the given query/sentence as an input and returns list of tokens which are lemmatized. Lemmatization refers to deriving the root word which is morphologically correct. There rises a slight confusion between lemmatization and 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 removal 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 or dictionary form of a word, which is known as the lemma.
Stopwords, are the high frequency words in a language which do not contribute much to the topic of the sentence. In English, such words include, ‘a’ , ‘an’ , ‘the’, ‘of’ , ‘to’ , etc.. We remove these words and focus on our main subject/topic, to solve ambiguity. The function is applied to the training data sets as well as user input.

```
def simlilarityCheck(word1, word2):

        word1 = word1 + ".n.01"
        word2 = word2 + ".n.01"
        try:
                w1 = wordnet.synset(word1)
                w2 = wordnet.synset(word2)

                return w1.wup_similarity(w2)

        except:
                return 0
```

Next we perform similarity check, function : similarityCheck, for the filtered sentence tokens that are returned by the first function. Similarity is checked between the given query/sentence tokens and the training data set tokens. For this, the synonym set is loaded for each token word from wordnet corpus. The depth and closeness of a word is calculated and returned on scale of 0–1 . This is the main data that will resolve the ambiguity. The more data you provide, the more accurate it gets. The normalized similarity between sentences is stored.

```
def synonymsCreator(word):
        synonyms = []

        for syn in wordnet.synsets(word):
                for i in syn.lemmas():
                        synonyms.append(i.name())

        return synonyms
```

synonymsCreator is a simplistic function to store the synonyms of the given input word. This will be used is storing the synonyms of the given data set and query tokens. The synonyms will also be taken into consideration while performing similarity check for the sentences.

```
# Remove Stop Words . Word Stemming . Return new tokenised list.
def filteredSentence(sentence):

        filtered_sent = []
        lemmatizer = WordNetLemmatizer()   #lemmatizes the words
        ps = PorterStemmer()    #stemmer stems the root of the word.

        stop_words = set(stopwords.words("english"))
        words = word_tokenize(sentence)

        for w in words:
        if w not in stop_words:
```

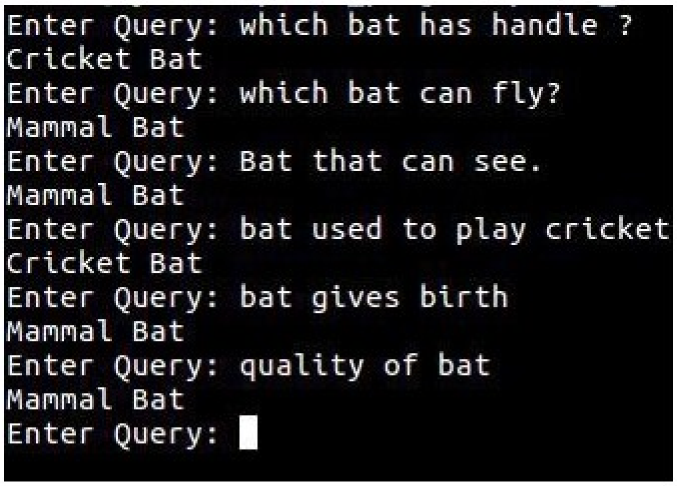
```
			filtered_sent.append(lemmatizer.lemmatize(ps.stem(w)))
			for i in synonymsCreator(w):
				filtered_sent.append(i)
		return filtered_sent
```

Once the similarity is stored, we apply the next level filter, function: filteredSentence , to apply lemmatization over stemmed tokens and again removing stop words. In the filtered sentence list, we now store the token word along with its synonyms for more precised matching / similarity check. Next, we put all these together.

```
if __name__ == '__main__':

	cricfile = codecs.open("cricketbat.txt", 'r', "utf-8")
	sent2 = cricfile.read().lower()
	vampirefile = codecs.open("vampirebat.txt", 'r', 'utf-8')
	sent1 = vampirefile.read().lower()
	sent3 = "start"

	# FOR TEST , replace the above variables with below sent1 and sent 2
	# sent1 = "the commercial banks are used for finance. all the financial matters are managed
by financial banks and they have lots of money, user accounts like salary account and savings
account, current account. money can also be withdrawn from this bank."
	# sent2 = "the river bank has water in it and it has fishes trees . lots of water is stored in the
banks. boats float in it and animals come and drink water from it."
	# sent3 = "from which bank should i withdraw money"

	while(sent3 != "end"):

		sent3 = raw_input("Enter Query: ").lower()

		filtered_sent1 = []
		filtered_sent2 = []
		filtered_sent3 = []

		counter1 = 0
		counter2 = 0
		sent31_similarity = 0
		sent32_similarity = 0

		filtered_sent1 = simpleFilter(sent1)
		filtered_sent2 = simpleFilter(sent2)
		filtered_sent3 = simpleFilter(sent3)

		for i in filtered_sent3:

			for j in filtered_sent1:
				counter1 = counter1 + 1
				sent31_similarity = sent31_similarity + simlilarityCheck(i,j)

			for j in filtered_sent2:
				counter2 = counter2 + 1
```

```
                                        sent32_similarity = sent32_similarity + simlilarityCheck(i,j)

                        filtered_sent1 = []
                        filtered_sent2 = []
                        filtered_sent3 = []

                        filtered_sent1 = filteredSentence(sent1)
                        filtered_sent2 = filteredSentence(sent2)
                        filtered_sent3 = filteredSentence(sent3)

                        sent1_count = 0
                        sent2_count = 0

                        for i in filtered_sent3:

                                for j in filtered_sent1:

                                        if(i==j):
                                                sent1_count = sent1_count + 1

                                for j in filtered_sent2:
                                        if(i==j):
                                                sent2_count = sent2_count + 1

                        if((sent1_count + sent31_similarity)>(sent2_count+sent32_similarity)):
                                print "Mammal Bat"
                        else:
                                print "Cricket Bat"

                        #---------------------------------------------
                        #Sentence1: the river bank has water in it and it has fishes trees . lots of water is
stored in the banks. boats float in it and animals come and drink water from it.
                        #sentence2: the commercial banks are used for finance. all the financial matters
are managed by financial banks and they have lots of money, user accounts like salary account and
savings account, current account. money can also be withdrawn from this bank.
                        #query: from which bank should i withdraw money.

                        #sen1: any of various nocturnal flying mammals of the order Chiroptera, having
membranous wings that extend from the forelimbs to the hind limbs or tail and anatomical
adaptations for echolocation, by which they navigate and hunt prey.
                        #sen 2: a cricket wooden bat is used for playing criket. it is rectangular in shape
and has handle and is made of wood or plastic and is used by cricket players.
                print "\nTERMINATED"
```

What done here, is, the application has been fed with two data set files, first cricketbat.txt , which contains few sentences referring to bat used in cricket sport, and second, vampirebat.txt, which contains few sentences referring to the mammal bird bat. sent1 stores the lowered case string data from the vampirebat.txt file and sent2 does for cricketbat.txt, sent3 stores user query. Next, the sentences are filtered and similarity is checked using the functions explained above. The comparison is normalized, and output is given accordingly whether the query refers to Cricket bat or Mammal bat.



```
Enter Query: which bat has handle ?
Cricket Bat
Enter Query: which bat can fly?
Mammal Bat
Enter Query: Bat that can see.
Mammal Bat
Enter Query: bat used to play cricket
Cricket Bat
Enter Query: bat gives birth
Mammal Bat
Enter Query: quality of bat
Mammal Bat
Enter Query: 
```

The program gives quiet accurate answers. The only thing it cannot handle are the negation sentences, like “which bat is not used to play cricket” , “which bat does not fly” , etc.
The data set and code has been included in the https://github.com/omkar-dsd/mini_projects/tree/master/word_sense_disambiuation

Information 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 retrieval* might be defined thus:

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 professional 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. Information retrieval is fast becoming the dominant form of information access, overtaking traditional database-style searching. 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 maintain 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 represented in documents by explicit markup (such as the coding underlying web pages). IR is also used to facilitate "semi structured" 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 documents. 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 arranging 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 manipulating 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 web char link. 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 classification: 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 handling 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 typically be stored on centralized file systems and one or a handful of dedicated machines will provide search over the collection.

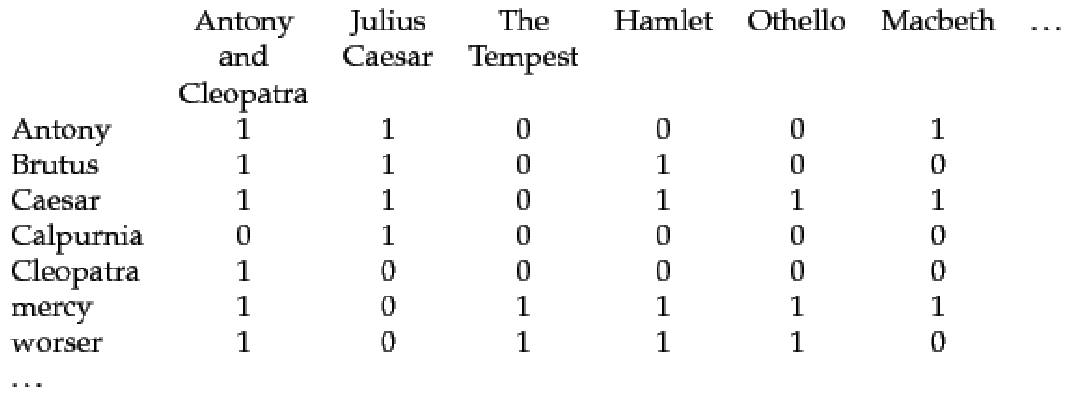
Term-document Incidence:

A fat book which many people own is Shakespeare's Collected Works. Suppose 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 contains Calpurnia. The simplest form of document retrieval is for a computer to do this sort of linear scan through documents. This process is commonly 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 . 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.

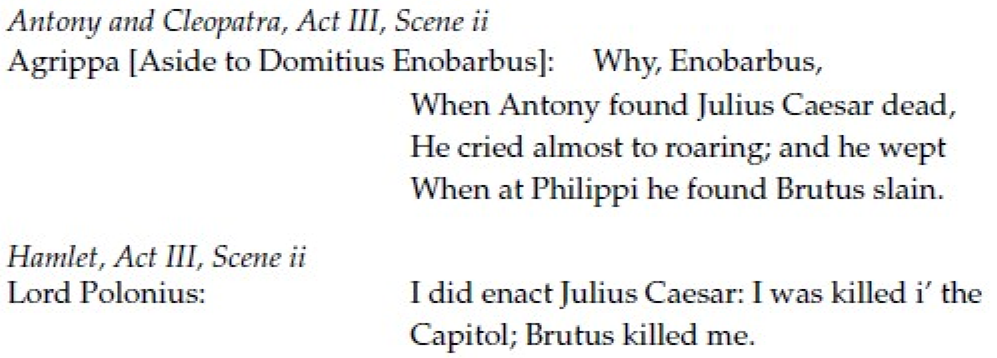
The way to avoid linearly scanning the texts for each query is to index the documents in advance. Let us stick with Shakespeare's Collected Works, and use it to introduce the basics of the Boolean retrieval model. Suppose we record for each document - here a play of Shakespeare's - whether it contains each word out of all the words Shakespeare used (Shakespeare used about 32,000 different words). The result is a binary term-document incidence matrix , as in Figure 1.1 . Terms are the indexed units; they are usually words, and for the moment you can think of 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.



	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
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.

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 (Figure 1.2).
The Boolean retrieval model is a model for information retrieval in which we 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.



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.

Let us now consider a more realistic scenario, simultaneously using the opportunity to introduce some terminology and notation. Suppose we have N = 1million 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 . We will refer to the group of documents over which we perform retrieval as the (document) collection . It is sometimes also referred to as a corpus (a body of texts). Suppose each document is about 1000 words long (2-3 book pages). If 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 = 500000 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.
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 is the topic about which the user desires to know more, and is differentiated from a query , which is what the user conveys to the computer in an attempt to communicate the information need. A document is 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 particular 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 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 : What fraction of the returned results are relevant to the information need?
Recall : What fraction of the relevant documents in the collection were returned by the system?

We now cannot build a term-document matrix in a naive way. A 500K X 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 . The name is actually redundant: an index always maps back from terms to the parts of a document where they occur. Nevertheless, inverted index, or sometimes inverted file , has become the standard term in information retrieval. The basic idea of an inverted index is shown in Figure 1.3 . We keep a dictionary of terms (sometimes also referred to as a vocabulary or lexicon ; here, we use dictionary for the data structure and vocabulary 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 document) - is conventionally called a posting . The list is then called a postings list (or), and all the postings lists taken together are referred to as the postings . The dictionary in Figure 1.3 has been sorted alphabetically and each postings list is sorted by document ID.

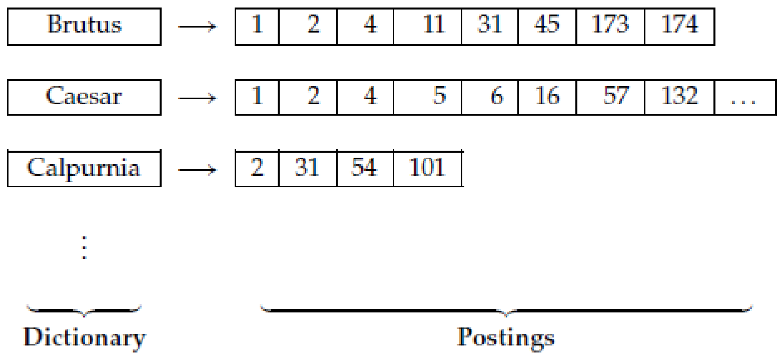


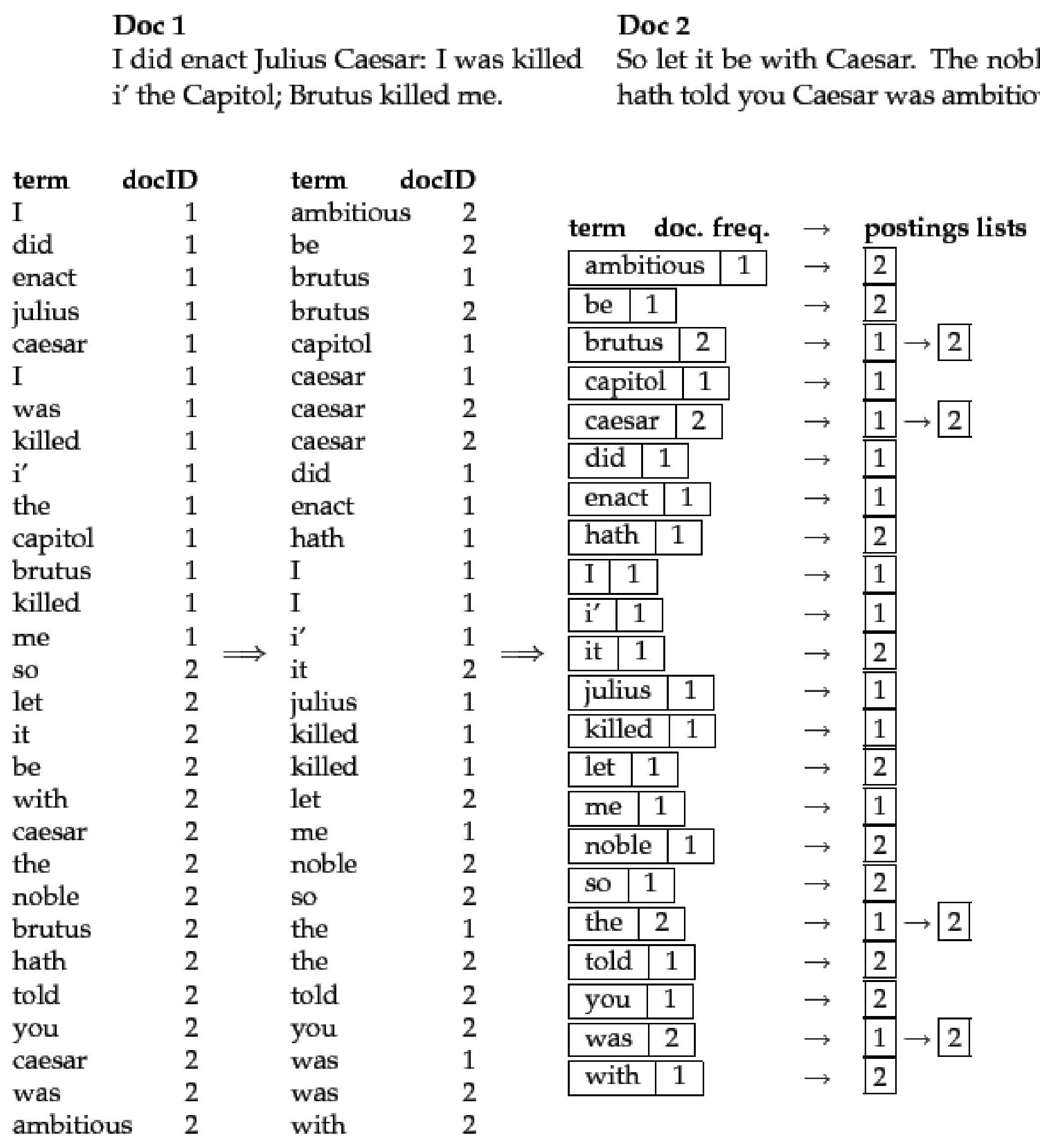
Figure 1.3 The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

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:
   $\boxed{\text{Friends, Romans, countrymen.}}$ $\boxed{\text{So let it be with Caesar}}$ ...
2. Tokenize the text, turning each document into a list of tokens:
   $\boxed{\text{Friends}}$ $\boxed{\text{Romans}}$ $\boxed{\text{countrymen}}$ $\boxed{\text{So}}$ ...
3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms: $\boxed{\text{friend}}$ $\boxed{\text{roman}}$ $\boxed{\text{countryman}}$ $\boxed{\text{so}}$ ...
4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

Here, we assume that the first 3 steps have already been done, and we examine building a basic inverted index by sort-based indexing .



Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
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

$\Longrightarrow$

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

$\Longrightarrow$

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 (*middle*). 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.

Within a document collection, we assume that each document has a unique serial number, known as the document identifier (docID). During index construction, 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 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. Instances 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 documents, this data organization already reduces the storage requirements of the index. The dictionary also records some statistics, such as the number of documents which contain each term (the document frequency , which is here also the length of each postings list). This information is not vital for a basic Boolean search engine, but it allows us to improve the efficiency of the search engine at query time, and it is a statistic later used in many ranked retrieval models. The postings are secondarily sorted by docID. This provides the basis for efficient query processing. This inverted index structure is essentially 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. 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 documents), and naturally extend to more advanced indexing strategies such as skip lists, which require additional pointers. Variable length arrays 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.

Query Optimization:
How do we process a query using an inverted index and the basic Boolean retrieval model? Consider processing the *simple conjunctive query*:
Brutus and Calpurnia
over the inverted index partially shown in Figure 1.3 .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.

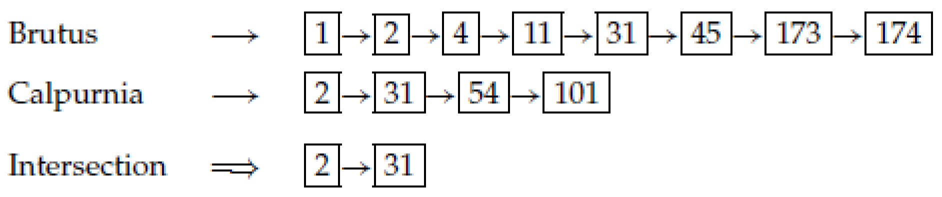


Figure 1.5 Intersecting the postings lists for Brutus and Calpurnia from Figure 1.3.

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 terms. (This operation is sometimes referred to as *merging* postings lists, this merge slightly counterintuitive name reflects using the term *merge algorithm* for a general family of algorithms that combine multiple sorted lists by interleaved 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 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 $\Theta(N)$, where N is the number of documents in the collection. Our indexing methods gain us just a constant, not a difference in Θ time complexity compared with 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:

(Brutus or Caesar) and not Calpurnia

Query optimization is the process of selecting how to organize the work of answering 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? Consider a query that is an and of t terms, for instance:

Brutus and Caesar and Calpurnia

```
INTERSECT($p_1$, $p_2$)
answer ← ⟨ ⟩
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
      then ADD(answer, $docID(p_1)$)
           $p_1 \leftarrow next(p_1)$
           $p_2 \leftarrow next(p_2)$
      else if $docID(p_1) < docID(p_2)$
             then $p_1 \leftarrow next(p_1)$
             else $p_2 \leftarrow next(p_2)$
return answer
```

Figure 1.6 Algorithm for the intersection of two postings lists p_1 and p_2.

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 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, we execute the above query as:

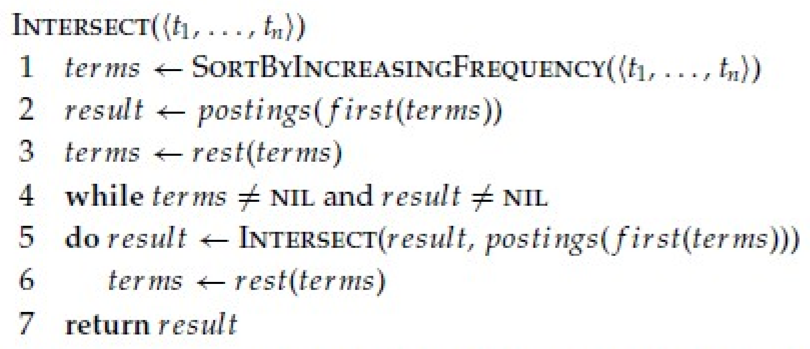
(Calpurnia and Brutus) and Caesar

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:

(madding or crowd) and (ignoble or strife) and (killed or slain)

As before, we get the frequencies for all terms, and we can then (conservatively) 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 postings lists as a function with two inputs and a distinct output, it is more efficient 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 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 alternative 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 hash table, so that membership of an intermediate result item can be calculated in constant rather than linear or log time. Moreover, standard postings list intersection operations remain necessary when both terms of a query are very common.



```
INTERSECT($\langle t_1, \ldots, t_n \rangle$)
terms ← SORTBYINCREASINGFREQUENCY($\langle t_1, \ldots, t_n \rangle$)
result ← postings(first(terms))
terms ← rest(terms)
while terms ≠ NIL and result ≠ NIL
do result ← INTERSECT(result, postings(first(terms)))
    terms ← rest(terms)
return result
```

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

Ranked Retrieval:

The Boolean retrieval model contrasts with *ranked retrieval models* such as the vector space model , in which users largely use *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 research on the advantages of ranked retrieval, systems implementing the Boolean retrieval model were the main or only search option provided by large commercial information providers for three decades until the early 1990s (approximately the date of arrival of the World Wide Web). However, these systems did not have just the basic Boolean operations (and, or, and not) that have been 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 operator. A *proximity operator* is a way of specifying that two terms in a operator query must occur close to each other in a document, where closeness may be measured 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 subscribers), 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:

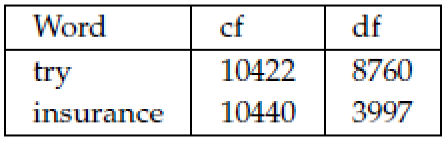
Information need: Information on the legal theories involved in preventing
the disclosure of trade secrets by employees formerly employed by a competing company.
Query: “trade secret” /s disclos! /s prevent /s employe!
Information need: Requirements for disabled people to be able to access a workplace.
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 binding 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);. The exclamation mark (!) gives a trailing wildcard query ; thus liab! matches all words starting with liab. Additionally work-site matches any of

worksite, *work-site* or *work site*. Typical expert queries are usually carefully defined and incrementally 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 offers the user greater control and transparency over what is retrieved. And some domains, 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 recommend 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, experimenting on a Westlaw sub collection, 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 precision 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.

Term Frequency and Inverse Document Frequency based Ranking:

Thus far, we have dealt with indexes that support Boolean queries: A document either matches or does not match a query. In the case of large document collections, the resulting number of matching documents can far exceed the
number a human user could possibly sift through. Accordingly, it is essential for a search engine to rank-order the documents matching a query. To do this, the search engine computes, for each matching document, a score with
respect to the query at hand. Here, we initiate the study of assigning a score to a (query, document) pair.
We assign to each term in a document a *weight* for that term that depends on the number of occurrences of the term in the document. We would like to compute a score between a query term *t* and a document *d*, based on the weight of *t* in *d*. The simplest approach is to assign the weight to be equal to the number of occurrences of term *t* in document *d*. This weighting scheme is referred term to as *term frequency* and is denoted tf*t*,*d* , with the subscripts denoting the term and the document in order.
For a document *d*, the set of weights determined by the tf weights above (or indeed any weighting function that maps the number of occurrences of *t* in *d* to a positive real value) may be viewed as a quantitative digest of that
document. In this view of a document, known in the literature as the *bag of words model*, the exact ordering of the terms in a document is ignored but the number of occurrences of each term is material (in contrast with Boolean
retrieval). We only retain information on the number of occurrences of each term. Thus, the document *Mary is quicker than John* is, in this view, identical to the document *John is quicker than Mary*. Nevertheless, it seems intuitive that two documents with similar bag of words representations are similar in content.



Word	cf	df
try	10422	8760
insurance	10440	3997

Figure 6.7 Collection frequency (cf) and document frequency (df) behave differently, as in this example from the Reuters-RCV1 collection.

Raw term frequency as above suffers from a critical problem: All terms are considered equally important when it comes to assessing relevancy on a query. In fact, certain terms have little or no discriminating power in determining
relevance. For instance, a collection of documents on the auto industry is likely to have the term auto in almost every document. To this end, we introduce a mechanism for attenuating the effect of terms that occur too often in the collection to be meaningful for relevance determination. An immediate idea is to scale down the term weights of terms with high *collection frequency*, defined to be the total number of occurrences of a term in the collection. The idea is to reduce the tf weight of a term by a factor that grows with its collection frequency.

Instead, it is more commonplace to use for this purpose the *document frequency* df*t* , defined to be the number of documents in the collection that contain a term *t*. This is because in trying to discriminate between documents
for the purpose of scoring, it is better to use a document-level statistic (such as the number of documents containing a term) than to use a collection-wide statistic for the term. The reason to prefer df to cf is illustrated in Figure 6.7,
where a simple example shows that collection frequency (cf) and document frequency (df) can behave rather differently. In particular, the cf values for both try and insurance are roughly equal, but their df values differ significantly. Intuitively, we want the few documents that contain insurance to get a higher boost for a query on insurance than the many documents containing try get from a query on try.
How is the document frequency df of a term used to scale its weight? Denoting as usual the total number of documents in a collection by *N*, we define the *inverse document frequency* (idf) of a term *t* as follows:

$$\mathrm{idf}_t = \log \frac{N}{\mathrm{df}_t}.$$

term	df_t	idf_t
car	18,165	1.65
auto	6723	2.08
insurance	19,241	1.62
best	25,235	1.5

Figure 6.8 Example of idf values. Here we give the idf's of terms with various frequencies in the Reuters collection of 806,791 documents.

Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low. Figure 6.8 gives an example of idfs in the Reuters-RCV1 collection of 806,791 documents; in this example, logarithms are to the base 10.
We now combine the definitions of term frequency and inverse document frequency to produce a composite weight for each term in each document. The *tf–idf* weighting scheme assigns to term *t* a weight in document *d* given
by

$$\text{tf-idf}_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t.$$

In other words, tf–idf*t,d* assigns to term *t* a weight in document *d* that is
1. highest when *t* occurs many times within a small number of documents (thus lending high discriminating power to those documents);

2. lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal);
3. lowest when the term occurs in virtually all documents.
At this point, we may view each document as a *vector* with one component corresponding to each term in the dictionary, together with a weight for each component that is given by (6.8). For dictionary terms that do not occur in a document, this weight is zero. This vector form will prove to be crucial to scoring and ranking;. As a first
step, we introduce the *overlap score measure*: The score of a document *d* is the sum, over all query terms, of the number of times each of the query terms occurs in *d*. We can refine this idea so that we add up not the number of
occurrences of each query term *t* in *d*, but instead the tf–idf weight of each term in *d*.

$$\text{Score}(q, d) = \sum_{t \in q} \text{tf–idf}_{t,d}.$$

Zone Indexing:
Digital documents generally encode, metadata in machine-recognizable form, certain *metadata* associated with each document. By metadata, we mean specific forms of data about a document, such as its author(s), title, and date of publication. These metadata would generally include *fields*, such as the date of creation and the format of the document, as well the author and possibly the title of the document. The possible values of a field should be thought of as finite – for instance, the set of all dates of authorship. Consider queries of the form "find documents authored by William Shakespeare in 1601, containing the phrase alas poor Yorick." Query processing then consists as usual of postings intersections, except that we may merge postings from standard inverted as well as *parametric indexes*. There is one parametric index for each field (say, date of creation); it allows us to select only the documents matching a date specified in the query. Figure 6.1 illustrates the user's view of such a parametric search. Some of the fields may assume ordered values, such as dates; in the example query above, the year 1601 is one such field value. The search engine may support querying ranges on such ordered values; to this end, a structure like a B-tree may be used for the field's dictionary.
Zones are similar to fields, except the contents of a zone can be arbitrary free text. Whereas a field may take on a relatively small set of values, a zone can be thought of as an arbitrary, unbounded amount of text. For instance,
document titles and abstracts are generally treated as zones. We may build a separate inverted index for each zone of a document, to support queries such as "find documents with merchant in the title and william in the author
list and the phrase gentle rain in the body." This has the effect of building an index that looks like Figure 6.2. Whereas the dictionary for a parametric index comes from a fixed vocabulary (the set of languages, or the set of dates), the dictionary for a zone index must structure whatever vocabulary stems from the text of that zone.

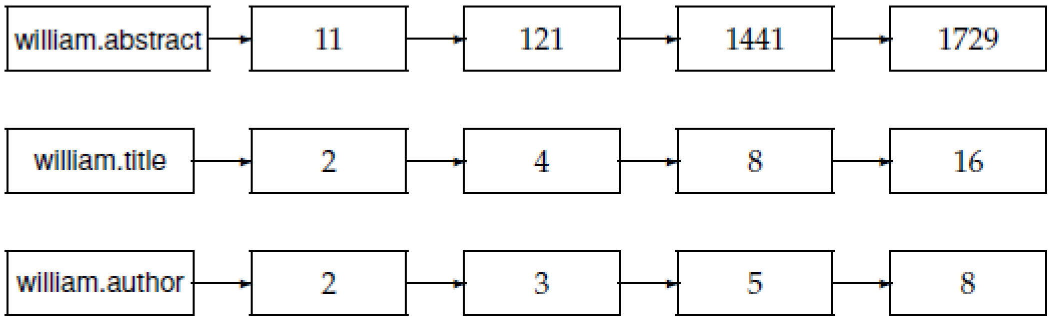


Figure 6.2 Basic zone index; zones are encoded as extensions of dictionary entries.

In fact, we can reduce the size of the dictionary by encoding the zone in which a term occurs in the postings. In Figure 6.3 for instance, we show how occurrences of william in the title and author zones of various documents
are encoded. Such an encoding is useful when the size of the dictionary is a concern (because we require the dictionary to fit in main memory). But there is another important reason why the encoding of Figure 6.3 is useful:
the efficient computation of scores using a technique we call *weighted zone*. Given a Boolean query *q* and a document *d*, weighted zone scoring assigns to the pair (*q*, *d*) a score in the interval [0, 1], by computing a linear
combination of *zone scores*, where each zone of the document contributes a Boolean value. More specifically, consider a set of documents, each of which

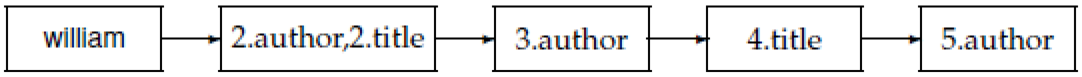


Figure 6.3 Zone index in which the zone is encoded in the postings rather than the dictionary.

has zones. Let $g1,....g \in [0, 1]$ such that $\sum_{i}^{l} g_i = 1$. For $1 \leq i \leq l$, let *si* be the Boolean score

denoting a match (or absence thereof) between *q* and the *i*th zone. For instance, the Boolean score from a zone could be 1 if all the
query term(s) occur in that zone, and zero otherwise; indeed, it could be any Boolean function that maps the presence of query terms in a zone to 0, 1. Then, the weighted zone score is defined to be

$$\sum_{i=1}^{\ell} g_i s_i .$$

Weighted zone scoring is sometimes referred to also as *ranked Boolean retrieval*.

Example 6.1: Consider the query shakespeare in a collection in which each document has three zones: *author*, *title*, and *body*. The Boolean score function for a zone takes on the value 1 if the query term shakespeare is present in the zone, and 0 otherwise. Weighted zone scoring in such a collection requires three weights *g*1, *g*2, and *g*3, respectively corresponding to the *author*, *title*, and *body* zones. Suppose we set *g*1 = 0.2, *g*2 = 0.3, and *g*3 = 0.5 (so that the three weights add up to 1); this corresponds to an application in which a match in the *author* zone is least important to the overall score, the *title* zone somewhat more, and the *body* contributes even more. Thus, if the term shakespeare were to appear in the *title* and *body* zones but not the *author* zone of a document, the score of this document would be 0.8.

Cosine Ranking:
The representation of a set of documents as vectors in a common vector space is known as the *vector space model* and is fundamental to a host of information retrieval model (IR) operations including scoring documents on a query, document classification, and document clustering. We first develop the basic ideas underlying vector space scoring; a pivotal step in this development is the view of queries as vectors in the same vector space as the document collection. We denote by $\bar{V}(d)$ the vector derived from document *d*, with one component in the vector for each dictionary term. Unless otherwise specified, the reader may assume that the components are computed using the tf–idf weighting scheme, although the particular weighting scheme is immaterial to the discussion that follows. The set of documents in a collection then may be viewed as a set of vectors in a vector space, in which there is one axis for each term. This representation loses the relative ordering of the terms in each document. A first attempt might consider the magnitude of the vector difference between two document vectors. This measure suffers from a drawback: Two documents with very similar content can have a significant vector difference simply because one is much longer than the other. Thus, the relative distributions of terms may be identical in the two documents, but the absolute term frequencies of one may be far larger.
To compensate for the effect of document length, the standard way of quantifying the similarity between two documents *d*1 and *d*2 is to compute the *cosine similarity* of their vector representations $\bar{V}(d_1)$ and $\bar{V}(d_2)$

$$\text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|},$$

where the numerator represents the *dot product* (also known as the *inner product*) of the vectors $\bar{V}(d_1)$ and $\bar{V}(d_2)$, and the denominator is the product of their *Euclidean lengths*. The dot product $\bar{x}.\bar{y}$ of two vectors is defined as $\sum_{i=1}^{M} x_i y_i$

Let $\bar{V}(d)$ denote the document vector for *d*, with *M* components $\bar{V}_1(d)...\bar{V}_m(d)$
The Euclidean length of *d* is defined as
$$\sqrt{\sum_{i=1}^{M} V^2(d)}$$

Search Engine Evaluation:
To measure ad hoc IR effectiveness in the standard way, we need a test collection consisting of three things:
1. A document collection

2. A test suite of information needs, expressible as queries
3. A set of relevance judgments, standardly a binary assessment of either *relevant* or *non relevant* for each query–document pair.

The standard approach to IR system evaluation revolves around the notion of *relevant* and *non relevant* documents. With respect to a user information need, a document in the test collection is given a binary classification as either relevant or non relevant. This decision is referred to as the *gold standard* or *ground truth* judgment of relevance. The test document collection and of information needs have to be of a reasonable size: You need to average performance over fairly large test sets because results are highly variable over different documents and information needs. As a rule of thumb, fifty information needs has usually been found to be a sufficient minimum. Relevance is assessed relative to an information need, *not* a query. For example, an information need might be:

Information on whether drinking red wine is more effective at reducing your risk of heart attacks than drinking white wine. This might be translated into a query such as:

wine and red and white and heart and attack and effective

A document is relevant if it addresses the stated information need, not because it just happens to contain all the words in the query. This distinction is often misunderstood in practice, because the information need is not overt.

But, nevertheless, an information need is present. If a user types python into a web search engine, they might be wanting to know where they can purchase a pet python. Or they might be wanting information on the programming language Python. From a one-word query, it is very difficult for a system to know what the information need is. But, nevertheless, the user has one, and can judge the returned results on the basis of their relevance to it. To evaluate a system, we require an overt expression of an information need, which can be used for judging returned documents as relevant or non relevant. At this point, we make a simplification: Relevance can reasonably be thought of as a scale, with some documents highly relevant and others marginally so. But, for the moment, we use just a binary decision of relevance. Many systems contain various weights (often known as parameters) that can be adjusted to tune system performance. It is wrong to report results on a test collection that were obtained by tuning these parameters to maximize performance on that collection. That is because such tuning overstates the expected performance of the system, because the weights will be set to maximize performance on one particular set of queries rather than for a random sample of queries. In such cases, the correct procedure is to have one or more *development test collections*, and to tune the parameters on the development test collection. The tester then runs the system with those weights on the test collection and reports the results on that collection as an unbiased estimate of performance.

The two most frequent and basic measures for information retrieval effectiveness are precision and recall. These are first defined for the simple case where an IR system returns a set of documents for a query. We will see later how to extend these notions to ranked retrieval situations.
Precision (*P*) is the fraction of retrieved documents that are relevant.

$$\text{Precision} = \frac{\#(\text{relevant items retrieved})}{\#(\text{retrieved items})} = P(\text{relevant}|\text{retrieved}).$$

Recall (*R*) is the fraction of relevant documents that are retrieved.

$$\text{Recall} = \frac{\#(\text{relevant items retrieved})}{\#(\text{relevant items})} = P(\text{retrieved}|\text{relevant}).$$

These notions can be made clear by examining the following contingency table:

	relevant	nonrelevant
retrieved	true positives (tp)	false positives (fp)
not retrieved	false negatives (fn)	true negatives (tn)

Then

$$P = tp/(tp + fp)$$
$$R = tp/(tp + fn).$$

An obvious alternative that may occur to the reader is to judge an information retrieval system by its *accuracy*, that is, the fraction of its classifications that are correct. In terms of the contingency table above,
accuracy =$(tp + tn)/(tp + f p + f n + tn)$.
This seems plausible, because there are two actual classes, relevant and non relevant, and an IR system can be thought of as a two-class classifier that attempts to label them as such (it retrieves the subset of documents it believes to be relevant). This is precisely the effectiveness measure often used for evaluating machine-learning classification problems.
A single measure that trades off precision versus recall is the *F measure*, which is the weighted harmonic mean of precision and recall:

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R} \quad \text{where} \quad \beta^2 = \frac{1-\alpha}{\alpha}$$

where $\alpha \in [0, 1]$ and thus $\beta 2 \in [0, \infty]$. The default *balanced F measure* equally weights precision and recall, which means making $\alpha = 1/2$ or $\beta = 1$. It is commonly written as $F1$, which is short for $F\beta{=}1$, even though the formulation in terms of α more transparently exhibits the F measure as a weighted harmonic mean. When using $\beta = 1$, the formula on the right simplifies to:

$$F_{\beta=1} = \frac{2PR}{P+R}.$$

However, using an even weighting is not the only choice. Values of $\beta < 1$ emphasize precision, whereas values of$\beta > 1$ emphasize recall. For example, a value of $\beta = 3$ or $\beta = 5$ might be used if recall is to be emphasized. Recall, precision, and the F measure are inherently measures between 0 and 1, but they are also very commonly written as percentages, on a scale between 0 and 100.

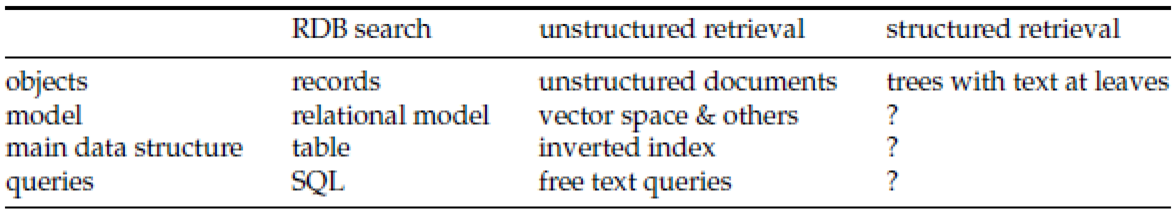
Resource Management with XML:

Information retrieval (IR) systems are often contrasted with relational databases. Traditionally, IR systems have retrieved information from *unstructured text* – by which we mean “raw” text without markup. Databases are
designed for querying *relational data*, sets of records that have values for predefined attributes such as employee number, title, and salary. There are fundamental differences between IR and database systems in terms of retrieval
model, data structures, and query language as shown in Table 10.1. Some highly structured text search problems are most efficiently handled by a relational database; for example, if the employee table contains an attribute for short textual job descriptions and you want to find all employees who are involved with invoicing. In this case, the SQL query:

```
select lastname from employees where job_desc like ’invoic%’;
```

may be sufficient to satisfy your information need with high precision and recall.

Table 10.1 Relational database (RDB) search, unstructured IR, and structured IR. There is no consensus yet as to which methods work best for structured retrieval, although many researchers believe that XQuery (page 197) will become the standard for structured queries.



	RDB search	unstructured retrieval	structured retrieval
objects	records	unstructured documents	trees with text at leaves
model	relational model	vector space & others	?
main data structure	table	inverted index	?
queries	SQL	free text queries	?

One standard for encoding structured documents: *EX tensible markup language* or *XML*, which is currently the most widely used such standard. We will not cover the specifics that distinguish XML from other types of markup such as HTML and SGML. But most of what we say in this chapter is applicable to markup languages in general. In the context of IR, we are only interested in XML as a language for encoding text and documents. A perhaps more widespread use of XML is to encode non text data. For example, we may want to export data in XML format from
an enterprise resource planning system and then read them into an analytics program to produce graphs for a presentation. This type of application of XML is called *data-centric* because numerical and non text attribute-value data XML dominate and text is usually a small fraction of the overall data. Most data centric XML is stored in databases – in contrast to the inverted index-based methods for text-centric XML that we present in this chapter.

We call XML retrieval *structured retrieval* in this chapter. Some researchers prefer the term *semistructured retrieval* to distinguish XML retrieval from retrieval database querying. We have adopted the terminology that is widespread in the XML retrieval community. For instance, the standard way of referring to XML queries is *structured queries*, not *semistructured queries*. The term *structured retrieval* is rarely used for database querying and it always refers to XML retrieval here.

```
<play>
<author>Shakespeare</author>
<title>Macbeth</title>
<act number="I">
<scene number="vii">
<title>Macbeth's castle</title>
<verse>Will I with wine and wassail ...</verse>
</scene>
</act>
</play>
```

Figure 10.1 An XML document.

An XML document is an ordered, labeled tree. Each node of the tree is an *XML element* and is written with an opening and closing *tag*. An element can XML have one or more *XML attributes*. In the XML document in Figure 10.1, the attribute *scene* element is enclosed by the two tags <scene ...> and </scene>. It has an attribute *number* with value *vii* and two child elements, *title* and *verse*.

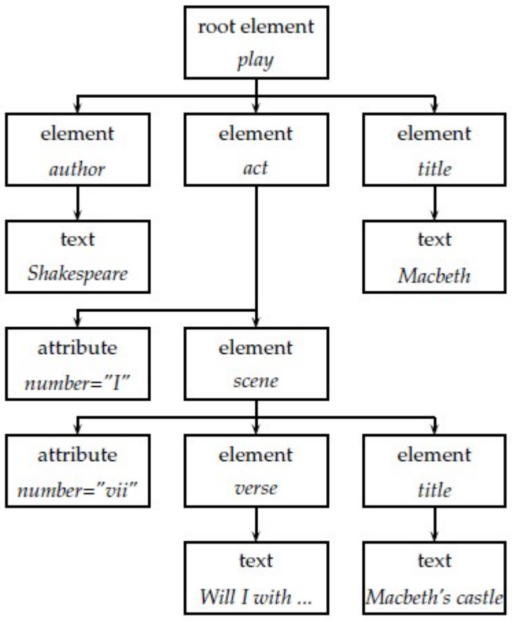


Figure 10.2 The XML document in Figure 10.1 as a simplified DOM object.

Figure 10.2 shows Figure 10.1 as a tree. The *leaf nodes* of the tree consist of text, for example, Shakespeare, Macbeth, and Macbeth's castle. The tree's *internal nodes* encode either the structure of the document (*title*, *act*, and *scene*) or metadata functions (*author*). The standard for accessing and processing XML documents is the XML DOM document object model or *DOM*. The DOM represents elements, attributes, and text within elements as nodes in a tree. Figure 10.2 is a simplified DOM representation of the XML document in Figure 10.1.With a DOM API, we can process an XML document by starting at the root element and then descending down the tree from parents to children. *XPath* is a standard for enumerating paths in an XML document collection. We will also refer to paths as *XML contexts* or simply *contexts* in this chapter. Only a small subset of XPath is needed for our purposes. The XPath expression node selects all nodes of that name. Successive elements of a path are separated by slashes, so act/scene selects all *scene* elements whose parent is an *act* element. Double slashes indicate that an arbitrary number of elements can intervene on a path: play//scene selects all *scene* elements occurring in a *play* element. In Figure 10.2, this set consists of a single *scene* element, which is accessible via the path *play*, *act*, *scene* from the top. An initial slash starts the path at the root element. /play/title selects the play's title in Figure 10.1, /play//title selects a set with two members (the play's title and the scene's title), and /scene/title selects no elements. For notational convenience, we allow the final element of a path to be a vocabulary term and separate it from the element path by the symbol #, even though this does not conform to the XPath standard. For example, title#"Macbeth" selects all titles containing the term Macbeth.

```
//article
[.//yr = 2001 or .//yr = 2002]
//section
[about(.,summer holidays)]
```

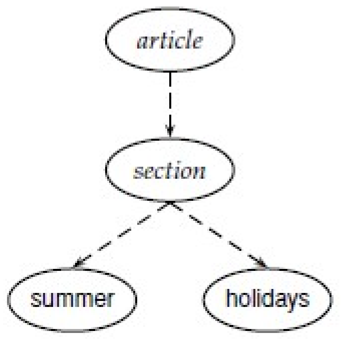


Figure 10.3 An XML query in NEXI format and its partial representation as a tree.

We also need the concept of *schema* in this chapter. A schema puts constraints on the structure of allowable XML documents for a particular application. A schema for Shakespeare's plays may stipulate that scenes can only occur as children of acts and that only acts and scenes have the *number* attribute. Two standards for schemas for XML documents are *XML DTD* XML schema (document type definition) and *XML schema*. Users can only write structured queries for an XML retrieval system if they have some minimal knowledge about the schema of the collection. A common format for XML queries is *NEXI* (Narrowed Extended XPath I). We give an example in Figure 10.3. We display the query on four lines for typographical convenience, but it is intended to be read as one unit without line breaks. In particular, //section is embedded under //article. The query in Figure 10.3 specifies a search for sections about the summer holidays that are part of articles from 2001 or 2002. As in XPath, double slashes indicate that an arbitrary number of elements can intervene on a path. The dot in a clause in square brackets refers to the element the clause modifies. The clause [.//yr = 2001 or .//yr = 2002] modifies //article. Thus, the dot refers to //article in this case. Similarly, the dot in [about(.,summer holidays)] refers to the section that the clause modifies.

The two yr conditions are relational attribute constraints. Only articles whose yr attribute is 2001 or 2002 (or that contain an element whose yr attribute is 2001 or 2002) are to be considered. The about clause is a ranking constraint: Sections that occur in the right type of article are to be ranked according to how relevant they are to the topic summer holidays. We usually handle relational attribute constraints by prefiltering or postfiltering: We simply exclude all elements from the result set that do not meet the relational attribute constraints. In this chapter, we will not address how to do this efficiently and instead focus on the core information retrieval problem in XML retrieval, namely, how to rank documents according to the relevance criteria expressed in the about conditions of the NEXI query. If we discard relational attributes, we can represent documents as trees with only

one type of node: element nodes. In other words, we remove all attribute nodes from the XML document, such as, the *number* attribute in
Figure 10.1. Figure 10.4 shows a subtree of the document in Figure 10.1 as an element–node tree (labeled *d*1).
We can represent queries as trees in the same way. This is a query-by example approach to query language design because users pose queries by creating objects that satisfy the same formal description as documents. In Figure
10.4, *q*1 is a search for books whose titles score highly for the keywords Julius Caesar. *q*2 is a search for books whose author elements score highly for Julius Caesar and whose title elements score highly for Gallic war.

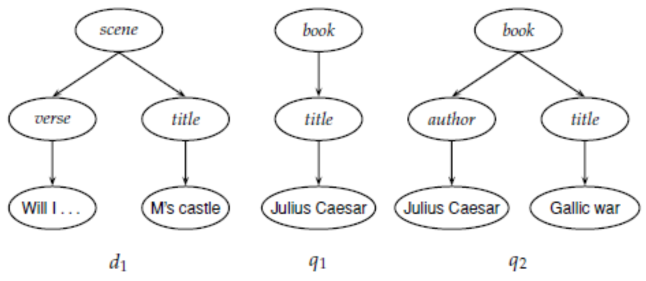


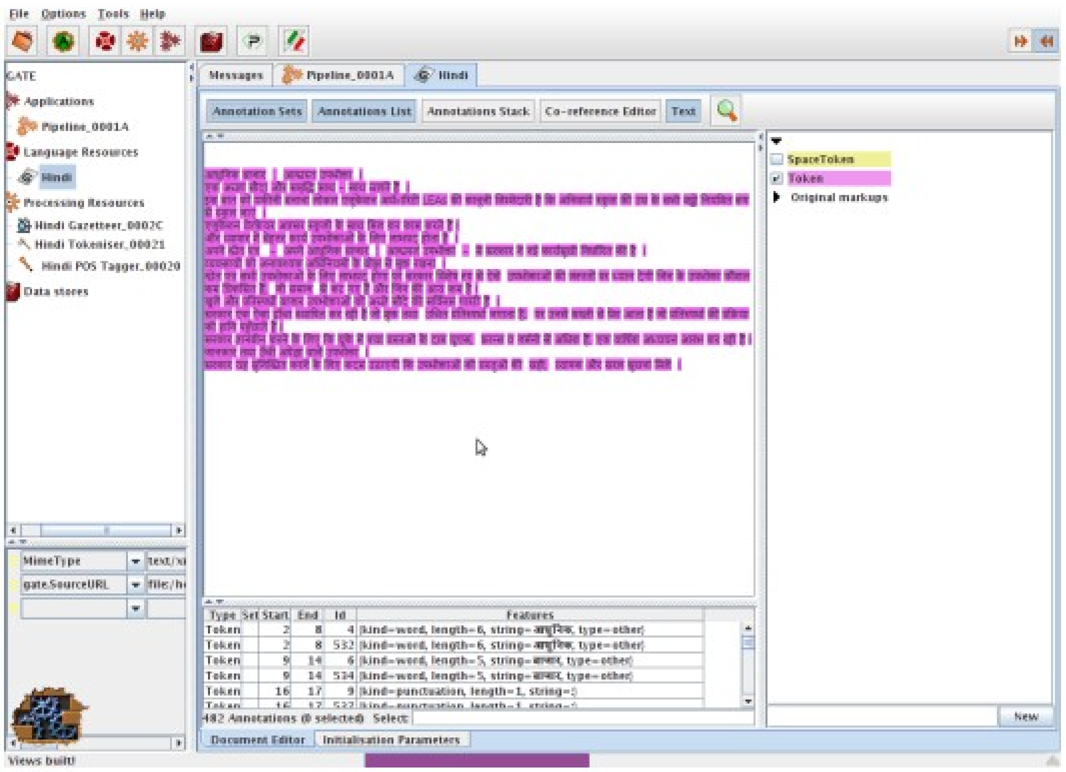
Figure 10.4 Tree representation of XML documents and queries.

For further studies please refer Introduction to Information Retrieval by C.D Manning.
GATE:
GATE is an infrastructure for developing and deploying software components that process human language. It is nearly 15 years old and is in active use for all types of computational task involving human language. GATE excels at text analysis of all shapes and sizes. From large corporations to small startups, from ¿multi-million research consortia to undergraduate projects, our user community is the largest and most diverse of any system of this type, and is spread across all but one of the continents. The GATE family of tools has grown over the years to include a desktop client for developers, a workflow-based web application, a Java library, an architecture and a process. GATE is:
ˆ (a) an IDE, GATE Developer: an integrated development environment for language processing components bundled with a very widely used Information Extraction system and a comprehensive set of other plug-ins.
ˆ (b) a cloud computing solution for hosted large-scale text processing, GATE Cloud (https://cloud.gate.ac.uk/).
(c) a web app, GATE Teamware: a collaborative annotation environment for factory style semantic annotation projects built around a workflow engine and a heavily optimized backend service infrastructure.
(d) a multi-paradigm search repository, GATE Mímir, which can be used to index and search over text, annotations, semantic schemas (ontologies), and semantic meta-data (instance data). It allows queries that arbitrarily mix full-text, structural, linguistic and semantic queries and that can scale to terabytes of text.
(e) a framework, GATE Embedded: an object library optimized for inclusion in diverse applications giving access to all the services used by GATE Developer and more.
(f) an architecture: a high-level organizational picture of how language processing software composition.
(g) a process for the creation of robust and maintainable services.

GATE as an architecture suggests that the elements of software systems that process natural language can usefully be broken down into various types of component, known as resources .

Components are reusable software chunks with well-defined interfaces, and are a popular architectural form, used in Sun's Java Beans and Microsoft's .Net, for example. GATE components are specialized types of Java Bean, and come in three flavours:
(a) LanguageResources (LRs) represent entities such as lexicons, corpora or ontologies;
(b) ProcessingResources (PRs) represent entities that are primarily algorithmic, such as parsers, generators or ngram modellers
(c) VisualResources (VRs) represent visualization and editing components that participate in GUIs.

Collectively, the set of resources integrated with GATE is known as CREOLE: a Collection of REusable Objects for Language Engineering. All the resources are packaged as Java Archive (or `JAR') les, plus some XML configuration data. The JAR and XML files are made available to GATE by putting them on a web server, or simply placing them in the local file space. When using GATE to develop language processing functionality for an application, the developer uses GATE Developer and GATE Embedded to construct resources of the three types. This may involve programming, or the development of Language Resources such as grammars that are used by existing Processing Resources, or a mixture of both. GATE
Developer is used for visualization of the data structures produced and consumed during processing, and for debugging, performance measurement and so on. For example, figure 1.1 is a screenshot of one of the visualization tools.



Type	Set	Start	End	Id	Features
Token		2	8	4	{kind=word, length=6, string=आधुनिक, type=other}
Token		2	8	532	{kind=word, length=6, string=आधुनिक, type=other}
Token		9	14	6	{kind=word, length=5, string=[illegible], type=other}
Token		9	14	534	{kind=word, length=5, string=[illegible], type=other}
Token		16	17	9	{kind=punctuation, length=1, string=।}
Token		16	17	532	{kind=punctuation, length=1, string=।}

482 Annotations (0 selected) Select:

Figure 1.1: One of GATE's visual resources

GATE Developer is analogous to systems like Mathematica for Mathematicians, or JBuilder for Java programmers: it provides a convenient graphical environment for research and development of language processing software. When an appropriate set of resources have been developed, they can then be embedded in the target client application using GATE Embedded. GATE Embedded is

supplied as a series of JAR files. To embed GATE-based language processing facilities in an application, these JAR files are all that is needed, along with JAR files and XML configuration files for the various resources that make up the new facilities. GATE includes resources for common LE data structures and algorithms, including documents, corpora and various annotation types, a set of language analysis components for
Information Extraction and a range of data visualization and editing components. GATE supports documents in a variety of formats including XML, RTF, email, HTML, SGML and plain text. In all cases the format is analyzed and converted into a single unified model of annotation. The annotation format is a modified form of the TIPSTER format [Grishman 97] which has been made largely compatible with the Atlas format [Bird & Liberman 99], and uses the now standard mechanism of `stand-omarkup'. GATE
documents, corpora and annotations are stored in databases of various sorts, visualized via the development environment, and accessed at code level via the framework. A family of Processing Resources for language analysis is included in the shape of ANNIE, A Nearly-New Information Extraction system. These components use finite state techniques to implement various tasks from tokenization to semantic tagging or verb phrase chunking. All ANNIE components communicate exclusively via GATE's document and annotation resources. for further resources please read https://gate.ac.uk/

NLTK(Natural Language Toolkit):
NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.
Thanks to a hands-on guide introducing programming fundamentals alongside topics in computational linguistics, plus comprehensive API documentation, NLTK is suitable for linguists, engineers, students, educators, researchers, and industry users alike. NLTK is available for Windows, Mac OS X, and Linux. Best of all, NLTK is a free, open source, community-driven project.

NLTK has been called “a wonderful tool for teaching, and working in, computational linguistics using Python,” and “an amazing library to play with natural language.”

Natural Language Processing with Python provides a practical introduction to programming for language processing. Written by the creators of NLTK, it guides the reader through the fundamentals of writing Python programs, working with corpora, categorizing text, analyzing linguistic structure, and more. The online version of the book has been updated for Python 3 and NLTK 3. For further resources please read https://www.nltk.org/

Reference:

a. D. Jurafsky & J. H. Martin – “Speech and Language Processing – An introduction to Language processing, Computational Linguistics, and Speech Recognition”,Pearson Education

b. Chris Manning and Hinrich Schütze, “Foundations of Statistical Natural Language Processing”, MIT Press. Cambridge, MA: May 1999.

c. Allen, James. 1995. – “Natural Language Understanding”. Benjamin/Cummings, 2ed.

d. Bharathi, A., Vineet Chaitanya and Rajeev Sangal. 1995. Natural Language Processing- “A Pananian Perspective”. Prentice Hll India, Eastern Economy Edition.

e. Siddiqui T., Tiwary U. S.. “Natural language processing and Information retrieval”, OUP, 2008.

f. Eugene Cherniak: “Statistical Language Learning”, MIT Press, 1993.

g. Manning, Christopher and Heinrich Schütze. 1999. “Foundations of Statistical Natural Language Processing”. MIT Press.