**Unit 1**

**Introduction to NLP:**

The field of natural language processing began in the 1940s, after World War II. At this time, people recognized the importance of translation from one language to another and hoped to create a machine that could do this sort of translation automatically. However, the task was obviously not as easy as people first imagined. By 1958, some researchers were identifying significant issues in the development of NLP. One of these researchers was Noam Chomsky, who found it troubling that models of language recognized sentences that were nonsense but grammatically correct as equally irrelevant as sentences that were nonsense and not grammatically correct. Chomsky found it problematic that the sentence “Colorless green ideas sleep furiously” was classified as improbable to the same extent that “Furiously sleep ideas green colorless”; any speaker of English can recognize the former as grammatically correct and the latter as incorrect, and Chomsky felt the same should be expected of machine models.

Around the same time in history, from 1957-1970, researchers split into two divisions concerning NLP: symbolic and stochastic. Symbolic, or rule-based, researchers focused on formal languages and generating syntax; this group consisted of many linguists and computer scientists who considered this branch the beginning of artificial intelligence research. Stochastic researchers were more interested in statistical and probabilistic methods of NLP, working on problems of optical character recognition and pattern recognition between texts.

After 1970, researchers split even further, embracing new areas of NLP as more technology and knowledge became available. One new area was logic-based paradigms, languages that focused on encoding rules and language in mathematical logics. This area of NLP research later contributed to the development of the programming language Prolog. Natural language understanding was another area of NLP that was particularly influenced by SHRDLU, Professor Terry Winograd’s doctoral thesis. This program placed a computer in a world of blocks, enabling it to manipulate and answer questions about the blocks according to natural language instructions from the user. The amazing part of this system was its capability to learn and understand with amazing accuracy, something only currently possible in extremely limited domains (e.g., the block world). The following text was generated in a demonstration of SHDRLU:



The computer is clearly able to resolve relationships between objects and understand certain ambiguities. A fourth area of NLP that came into existence after 1970 is discourse modeling. This area examines interchanges between people and computers, working out such ideas as the need to change “you” in a speaker’s question to “me” in the computer’s answer.

From 1983 to 1993, researchers became more united in focusing on empiricism and probabilistic models. Researchers were able to test certain arguments by Chomsky and others from the 1950s and 60s, discovering that many arguments that were convincing in text were not empirically accurate. Thus, by 1993, probabilistic and statistical methods of handling natural language processing were the most common types of models. In the last decade, NLP has also become more focused on information extraction and generation due to the vast amounts of information scattered across the Internet. Additionally, personal computers are now everywhere, and thus consumer level applications of NLP are much more common and an impetus for further research.

**Basics of Text Processing:**

**Words:**

Before we talk about processing words, we need to decide what counts as a word. corpus Let’s start by looking at one particular corpus (plural corpora), a computer-readable corpora collection of text or speech. For example the Brown corpus is a million-word collection of samples from 500 written English texts from different genres (newspaper, fiction, non-fiction, academic, etc.), assembled at Brown University in 1963–64 (Kuˇcera and Francis, 1967). How many words are in the following Brown sentence?

He stepped out into the hall, was delighted to encounter a water brother.

This sentence has 13 words if we don’t count punctuation marks as words, 15 if we count punctuation. Whether we treat period (“.”), comma (“,”), and so on as words depends on the task. Punctuation is critical for finding boundaries of things (commas, periods, colons) and for identifying some aspects of meaning (question marks, exclamation marks, quotation marks). For some tasks, like part-of-speech tagging or parsing or speech synthesis, we sometimes treat punctuation marks as if they were separate words.

The Switchboard corpus of American English telephone conversations between strangers was collected in the early 1990s; it contains 2430 conversations averaging 6 minutes each, totaling 240 hours of speech and about 3 million words (Godfrey et al., 1992). Such corpora of spoken language don’t have punctuation but do introduce other complications with regard to defining words. Let’s look at one utterance from Switchboard; an utterance is the spoken correlate of a sentence:

I do uh main- mainly business data processing

This utterance has two kinds of disfluencies. The broken-off word main- is called a fragment. Words like uh and um are called fillers or filled pauses. Should we consider these to be words? Again, it depends on the application. If we are building a speech transcription system, we might want to eventually strip out the disfluencies.

How about inflected forms like cats versus cat? These two words have the same **lemma** cat but are different **wordforms**. A lemma is a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense. The wordform is the full inflected or derived form of the word. For morphologically complex languages like Arabic, we often need to deal with lemmatization. For many tasks in English, however, wordforms are sufficient.

How many words are there in English? To answer this question we need to distinguish two ways of talking about words. **Types** are the number of distinct words in a corpus; if the set of words in the vocabulary is V, the number of types is the word token vocabulary size |V|. Tokens are the total number N of running words. If we ignore punctuation, the following Brown sentence has 16 tokens and 14 types:

**They picnicked by the pool, then lay back on the grass and looked at the stars**

Corpora

Words don’t appear out of nowhere. Any particular piece of text that we study is produced by one or more specific speakers or writers, in a specific dialect of a specific language, at a specific time, in a specific place, for a specific function**.**

It’s also quite common for speakers or writers to use multiple languages in a code switching single communicative act, a phenomenon called **code switching**. Code switching is enormously common across the world.

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. Tokenizing (segmenting) words
2. Normalizing word formats
3. Segmenting sentences

**Word Tokenization**

The tokenization is the task of segmenting running text into words.

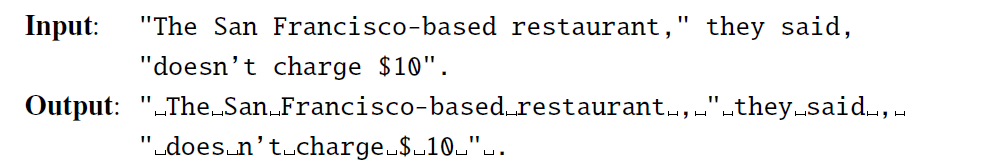
While the Unix command sequence just removed all the numbers and punctuation, for most NLP applications we’ll need to keep these in our tokenization. We often want to break off punctuation as a separate token; commas are a useful piece of information for parsers, periods help indicate sentence boundaries. But we’ll often want to keep the punctuation that occurs word internally, in examples like m.p.h., Ph.D., AT&T, and cap’n. Special characters and numbers will need to be kept in prices ($45.55) and dates (01/02/06); we don’t want to segment that price into separate tokens of “45” and “55”. And there are URLs (<https://www.stanford.edu>), Twitter hashtags (#nlproc), or email addresses ([someone@cs.colorado.edu](mailto:someone@cs.colorado.edu)).

Number expressions introduce other complications as well; while commas normally appear at word boundaries, commas are used inside numbers in English, every three digits: 555,500.50. Languages, and hence tokenization requirements, differ on this; many continental European languages like Spanish, French, and German, by contrast, use a comma to mark the decimal point, and spaces (or sometimes periods) where English puts commas, for example, 555 500,50.

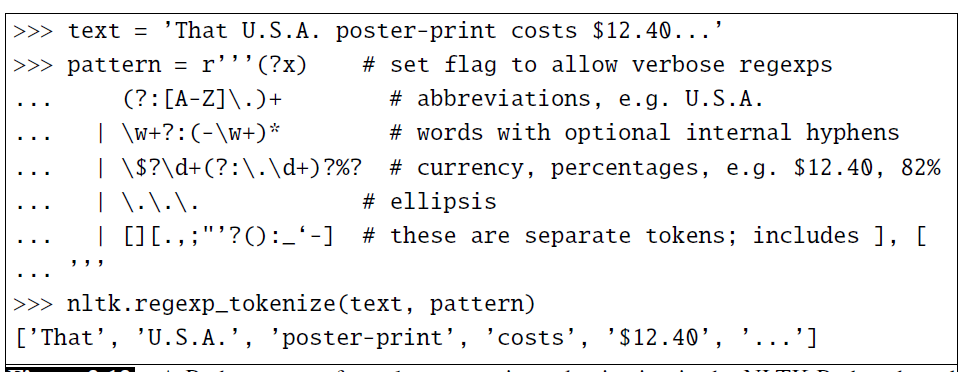
A tokenizer can also be used to expand **clitic** contractions that are marked by apostrophes, for example, converting what're to the two tokens what are, and we're to we are. A clitic is a part of a word that can’t stand on its own, and can only occur when it is attached to another word. Some such contractions occur in other alphabetic languages, including articles and pronouns in French (j'ai, l'homme).

Depending on the application, tokenization algorithms may also tokenize multiword expressions like New York or rock 'n' roll as a single token, which requires a multiword expression dictionary of some sort. Tokenization is thus intimately tied up with named entity recognition, the task of detecting names, dates, and organizations.

One commonly used tokenization standard is known as the Penn Treebank tokenization standard, used for the parsed corpora (treebanks) released by the Lintokenization guistic Data Consortium (LDC), the source of many useful datasets. This standard separates out clitics (doesn’t becomes does plus n’t), keeps hyphenated words together, and separates out all punctuation (to save space we’re showing visible spaces ‘ ’ between tokens, although newlines is a more common output):



In practice, since tokenization needs to be run before any other language processing, it needs to be very fast. The standard method for tokenization is therefore to use deterministic algorithms based on regular expressions compiled into very efficient finite state automata which we can implement using **nltk.regexptokenize** function of the Python-based **Natural Language Toolkit (NLTK).**



Carefully designed deterministic algorithms can deal with the ambiguities that arise, such as the fact that the apostrophe needs to be tokenized differently when used as a genitive marker (as in the book’s cover), a quotative as in ‘The other class’, she said, or in clitics like they’re.

**Byte-Pair Encoding for Tokenization**

There is a third option to tokenizing text. Instead of defining tokens as words (whether delimited by spaces or more complex algorithms), or as characters (as in Chinese), we can use our data to automatically tell us what the tokens should be. This is especially useful in dealing with unknown words, an important problem in language processing. As we will see in the next chapter, NLP algorithms often learn some facts about language from one corpus (a training corpus) and then use these facts to make decisions about a separate test corpus and its language. Thus if our training corpus contains, say the words low, new, newer, but not lower, then if the word lower appears in our test corpus, our system will not know what to do with it.

**Subword:**

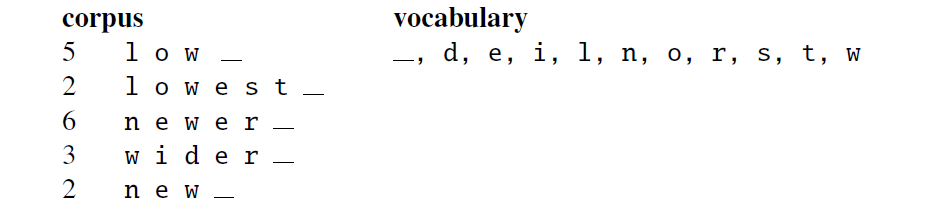
To deal with this unknown word problem, modern tokenizers often automatically induce sets of tokens that include tokens smaller than words, called **subwords.** Subwords can be arbitrary substrings, or they can be meaning-bearing units like the morphemes -est or -er. (A morpheme is the smallest meaning-bearing unit of a language; for example the word unlikeliest has the morphemes un-, likely, and -est.) In modern tokenization schemes, most tokens are words, but some tokens are frequently occurring morphemes or other subwords like -er. Every unseen word like lower can thus be represented by some sequence of known subword units, such as low and er, or even as a sequence of individual letters if necessary.

Most tokenization schemes have two parts: a token learner, and a token segmenter. The token learner takes a raw training corpus (sometimes roughly preseparated into words, for example by whitespace) and induces a vocabulary, a set of tokens. The token segmenter takes a raw test sentence and segments it into the tokens in the vocabulary. Three algorithms are widely used: byte-pair encoding (Sennrich et al., 2016), unigram language modeling (Kudo, 2018), and WordPiece (Schuster and Nakajima, 2012); there is also a SentencePiece library that includes implementations of the first two of the three (Kudo and Richardson, 2018a).

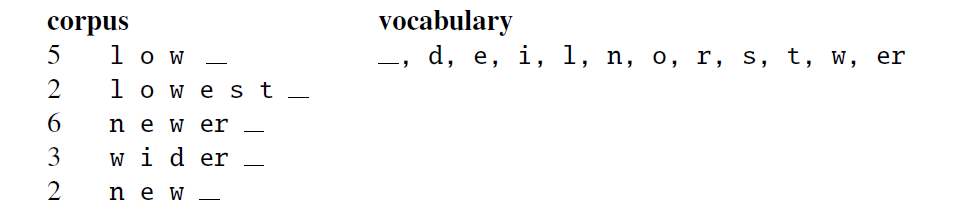
**Byte-pair Encoding:**

In this section we introduce the simplest of the three, the byte-pair encoding or BPE algorithm (Sennrich et al., 2016); see Fig. 2.13. The BPE token learner begins with a vocabulary that is just the set of all individual characters. It then examines the training corpus, chooses the two symbols that are most frequently adjacent (say ‘A’, ‘B’), adds a new merged symbol ‘AB’ to the vocabulary, and replaces every adjacent ’A’ ’B’ in the corpus with the new ‘AB’. It continues to count and merge, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus a parameter of the algorithm. The resulting vocabulary consists of the original set of characters plus k new symbols.

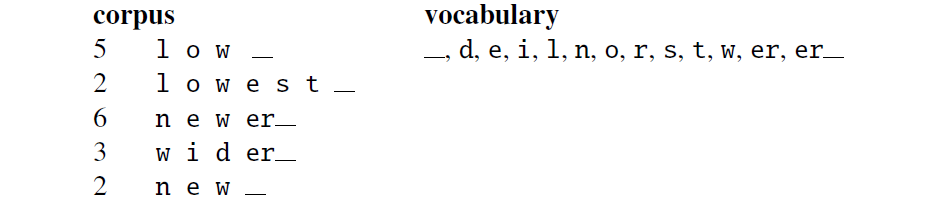
The algorithm is usually run inside words (not merging across word boundaries), so the input corpus is first white-space-separated to give a set of strings, each corresponding to the characters of a word, plus a special end-of-word symbol , and its counts. Let’s see its operation on the following tiny input corpus of 18 word tokens with counts for each word (the word low appears 5 times, the word newer 6 times, and so on), which would have a starting vocabulary of 11 letters:



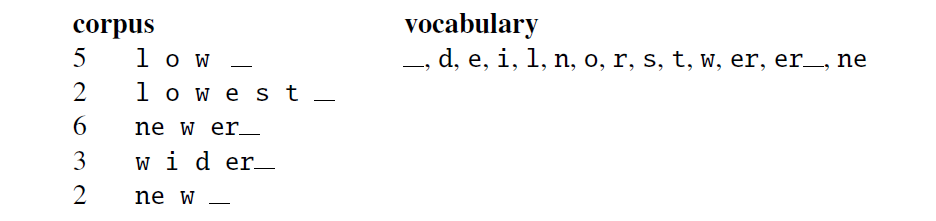
The BPE algorithm first counts all pairs of adjacent symbols: the most frequent is the pair e r because it occurs in newer (frequency of 6) and wider (frequency of 3) for a total of 9 occurrences.1 We then merge these symbols, treating er as one symbol, and count again:



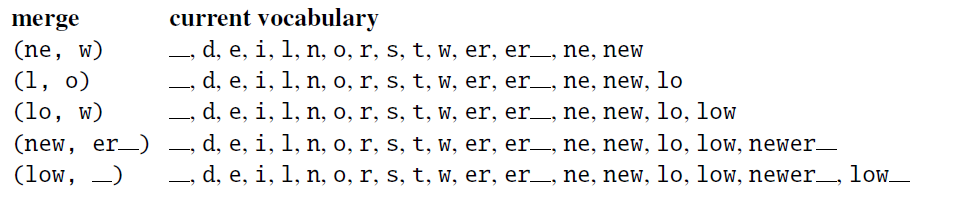
Now the most frequent pair is er\_ , which we merge; our system has learned that there should be a token for word-final er\_, represented as er\_ :

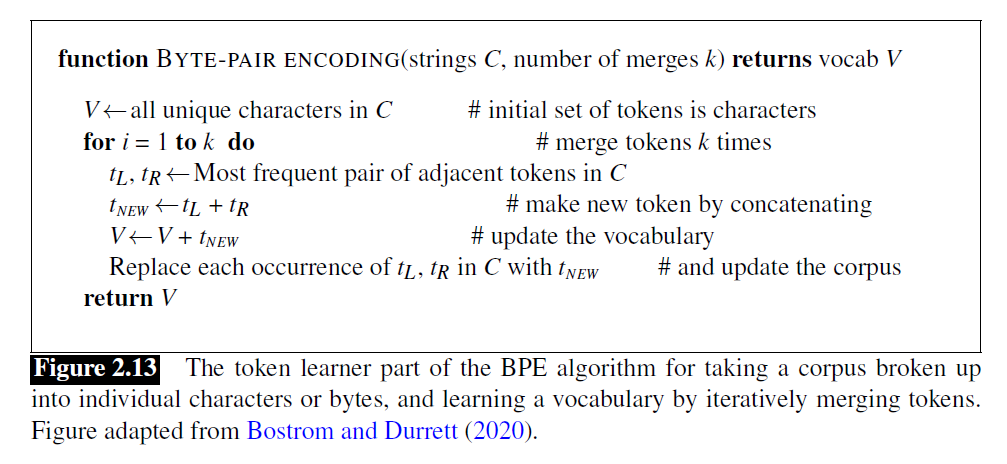


Next n e (total count of 8) get merged to ne:



If we continue, the next merges are:





Once we’ve learned our vocabulary, the token segmenter is used to tokenize a test sentence. The token segmenter just runs on the test data the merges we have learned from the training data, greedily, in the order we learned them. (Thus the frequencies in the test data don’t play a role, just the frequencies in the training data). So first we segment each test sentence word into characters. Then we apply the first rule: replace every instance of e r in the test corpus with er\_, and then the second rule: replace every instance of er in the test corpus with er , and so on. By the end, if the test corpus contained the character sequence **n e w e r** , it would be tokenized as a full word. But the characters of a new (unknown) word like **l o w e r** would be merged into the two tokens low er\_ .

**Word Normalization, Lemmatization Word normalization is the task of putting words/tokens in a standard format, choosing**

a single normal form for words with multiple forms like USA and US or uh-huh and uhhuh. This standardization may be valuable, despite the spelling information that is lost in the normalization process. For information retrieval or information and Stemming extraction about the US, we might want to see information from documents whether they mention the US or the USA.

**Case folding** is another kind of normalization. Mapping everything to lower case means that Woodchuck and woodchuck are represented identically, which is very helpful for generalization in many tasks, such as information retrieval or speech recognition. For sentiment analysis and other text classification tasks, information extraction, and machine translation, by contrast, case can be quite helpful and case folding is generally not done. This is because maintaining the difference between, for example, US the country and us the pronoun can outweigh the advantage in generalization that case folding would have provided for other words.

For many natural language processing situations we also want two morphologically different forms of a word to behave similarly. For example in web search, someone may type the string woodchucks but a useful system might want to also return pages that mention woodchuck with no s. This is especially common in morphologically complex languages like Polish, where for example the word Warsaw has different endings when it is the subject (Warszawa), or after a preposition like “in Warsaw” (w Warszawie), or “to Warsaw” (do Warszawy), and so on.

**Sentence Segmentation**

**What is a sentence?**

The first answer to what is a sentence is something ending with a ., ?’ or !’ .” We have already mentioned the problem that only some periods mark the end of a sentence: others are used to show an abbreviation, or for both these functions at once. Nevertheless, this basic heuristic gets one a long way: in general about 90% of periods are sentence boundary indicators (Riley 1989). There are a few other pitfalls to be aware of. Sometimes other punctuation marks split up what one might want to regard as a sentence. Often what is on one or the other or even both sides of the punctuation marks colon, semicolon, and dash (:, ;, and -) might best be thou.-.) might best be thought of as a sentence by itself, as .:’ in this example:

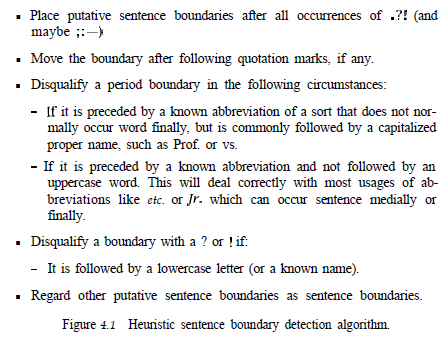
The scene is written with a combination of unbridled passion and surehanded control: In the exchanges of the three characters and the rise and fall of emotions, Mr. Weller has captured the heartbreaking inexorability of separation.

Related to this is the fact that sometimes sentences do not nicely follow in sequence, but seem to nest in awkward ways. While normally nested things are not seen as sentences by themselves, but clauses, this classification can be strained for cases such as the quoting of direct speech, where we get subsentences:

\*\*.ou remind me,\*\* she remarked, .\*\*f your mother.”

A second problem with such indirect speech is that it is standard typesetting practice (particularly in North America) to place quotation marks after sentence final punctuation. Therefore, the end of the sentence is not after the period in the example above, but after the close quotation mark that follows the period.

In practice most systems have used heuristic algorithms of this sort. With enough effort development, they can work very well, at least within the textual in their domain for which they were built. But any such solution suffers from the same problems of heuristic processes in other parts of the tokenization process. They require a lot of hand-coding and domain knowledge on the part of the person constructing the tokenizer, and tend to be brittle and domain-specific.



There has been increasing research recently on more principled methods of sentence boundary detection. Riley (1989) used statistical classification trees to determine sentence boundaries. The features for the classification trees include the case and length of the words preceding and following a period, and the a priori probability of different words to occur before and after a sentence boundary (the computation of which requires a large quantity of labeled training data). Palmer and Hearst

(1994; 1997) avoid the need for acquiring such data by simply using the part of speech distribution of the preceding and following words, and using a neural network to predict sentence boundaries. This yields a robust, largely language independent boundary detection algorithm with high performance (about 98-99% correct). Reynar and Ratnaparkhi (1997) and Mikheev (1998) develop Maximum Entropy approaches to the problem, the latter achieving an accuracy rate of 99.25% on sentence boundary prediction.

**Minimum Edit Distance**

strings are. For example in spelling correction, the user typed some erroneous string—let’s say graffe–and we want to know what the user meant. The user probably intended a word that is similar to graffe. Among candidate similar words, the word giraffe, which differs by only one letter from graffe, seems intuitively to be more similar than, say grail or graf, which differ in more letters. Another example comes from coreference, the task of deciding whether two strings such as

the following refer to the same entity:

*Stanford President Marc Tessier-Lavigne*

*Stanford University President Marc Tessier-Lavigne*

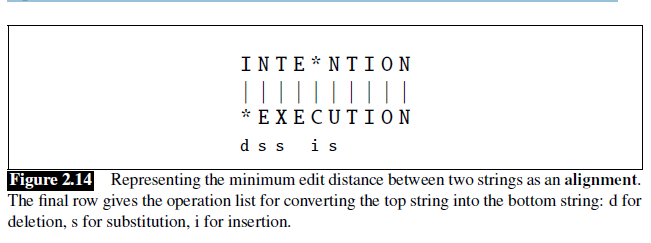
Again, the fact that these two strings are very similar (differing by only one word) seems like useful evidence for deciding that they might be coreferent.

**Edit distance** gives us a way to quantify both of these intuitions about string similarity. More formally, the minimum edit distance between two strings is defined as the minimum number of editing perations (operations like insertion, deletion, substitution) needed to transform one string into another.

The gap between intention and execution, for example, is 5 (delete an i, substitute e for n, substitute x for t, insert c, substitute u for n). It’s much easier to see this by looking at the most important visualization for string distances, an alignment between the two strings, shown in Fig. 2.14. Given two sequences, an alignment is a correspondence between substrings of the two sequences. Thus, we say I aligns with the empty string, N with E, and so on. Beneath the aligned strings is another

representation; a series of symbols expressing an operation list for converting the

top string into the bottom string: d for deletion, s for substitution, i for insertion.



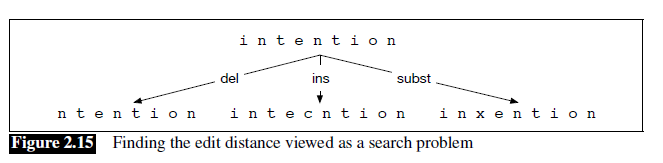
We can also assign a particular cost or weight to each of these operations. The Levenshtein distance between two sequences is the simplest weighting factor in which each of the three operations has a cost of 1 (Levenshtein, 1966)—we assume that the substitution of a letter for itself, for example, t for t, has zero cost. The Levenshtein distance between intention and execution is 5. Levenshtein also proposed an alternative version of his metric in which each insertion or deletion has a cost of

1 and substitutions are not allowed. (This is equivalent to allowing substitution, but giving each substitution a cost of 2 since any substitution can be represented by one insertion and one deletion). Using this version, the Levenshtein distance between intention and execution is 8.

**The Minimum Edit Distance Algorithm**

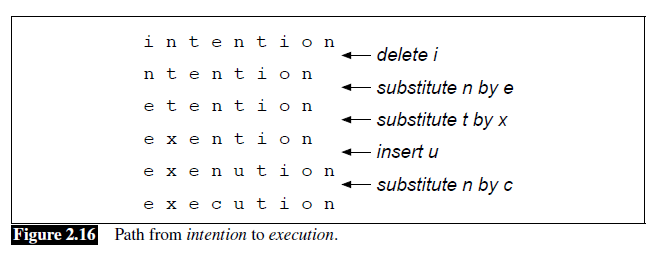
How do we find the minimum edit distance? We can think of this as a search task, in which we are searching for the shortest path—a sequence of edits—from one string to another.

The space of all possible edits is enormous, so we can’t search naively. However, lots of distinct edit paths will end up in the same state (string), so rather than recomputing all those paths, we could just remember the shortest path to a state each time we saw it. We can do this by using dynamic programming. Dynamic program ming is the name for a class of algorithms, first introduced by Bellman (1957), that



apply a table-driven method to solve problems by combining solutions to subproblems. Some of the most commonly used algorithms in natural language processing make use of dynamic programming, such as the Viterbi algorithm and the CKY algorithm for parsing.

The intuition of a dynamic programming problem is that a large problem can be solved by properly combining the solutions to various subproblems. Consider the shortest path of transformed words that represents the minimum edit distance between the strings intention and execution shown in Fig. 2.16.



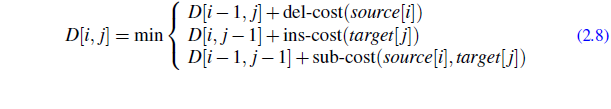
Imagine some string (perhaps it is exention) that is in this optimal path (whatever it is). The intuition of dynamic programming is that if exention is in the optimal operation list, then the optimal sequence must also include the optimal path from intention to exention. Why? If there were a shorter path from intention to exention, then we could use it instead, resulting in a shorter overall path, and the optimal

sequence wouldn’t be optimal, thus leading to a contradiction.

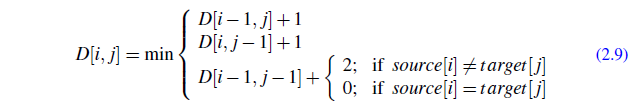
The minimum edit distance algorithm was named by Wagner and Fischer (1974) but independently discovered by many people

Let’s first define the minimum edit distance between two strings. Given two strings, the source string X of length n, and target string Y of length m, we’ll define D[i; j] as the edit distance between X[1::i] and Y[1:: j], i.e., the first i characters of X and the first j characters of Y. The edit distance between X and Y is thus D[n;m].

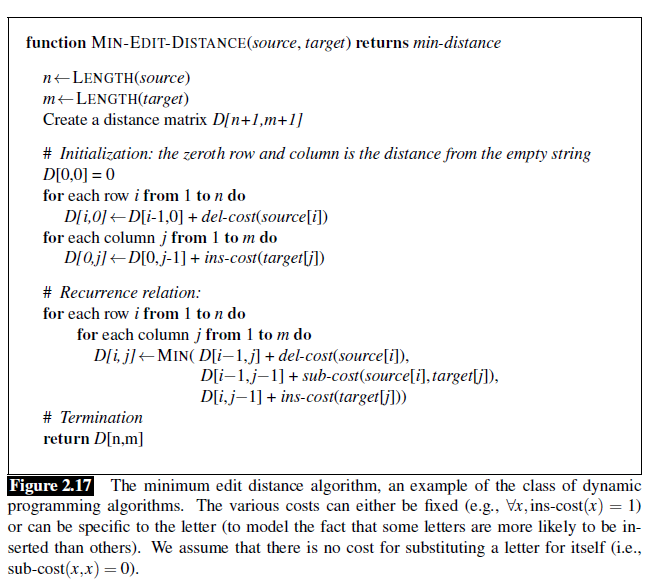
We’ll use dynamic programming to compute D[n;m] bottom up, combining solutions to subproblems. In the base case, with a source substring of length i but an empty target string, going from i characters to 0 requires i deletes. With a target substring of length j but an empty source going from 0 characters to j characters requires j inserts. Having computed D[i; j] for small i; j we then compute larger D[i; j] based on previously computed smaller values. The value of D[i; j] is computed by taking the minimum of the three possible paths through the matrix which arrive there:



If we assume the version of Levenshtein distance in which the insertions and deletions each have a cost of 1 (ins-cost(.) = del-cost(.) = 1), and substitutions have a cost of 2 (except substitution of identical letters have zero cost), the computation for D[i; j] becomes:

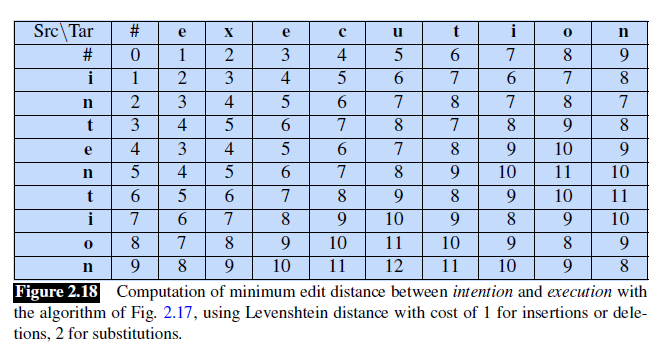


The algorithm is summarized in Fig. 2.17; Fig. 2.18 shows the results of applying the algorithm to the distance between intention and execution with the version of Levenshtein in Eq. 2.9.



Alignment Knowing the minimum edit distance is useful for algorithms like finding potential spelling error corrections. But the edit distance algorithm is important in another way; with a small change, it can also provide the minimum cost alignment between two strings. Aligning two strings is useful throughout speech and language processing. In speech recognition, minimum edit distance alignment is used to compute the word error rate (Chapter 16). Alignment plays a role in machine translation, in which sentences in a parallel corpus (a corpus with a text in two languages) need to be matched to each other.

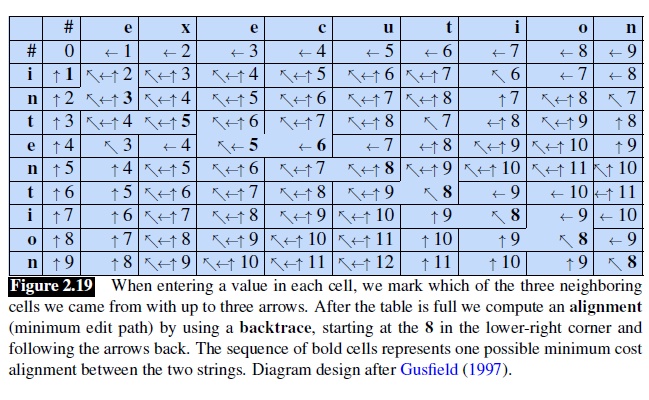
To extend the edit distance algorithm to produce an alignment, we can start by visualizing an alignment as a path through the edit distance matrix. Figure 2.19 shows this path with boldfaced cells. Each boldfaced cell represents an alignment



of a pair of letters in the two strings. If two boldfaced cells occur in the same row, there will be an insertion in going from the source to the target; two boldfaced cells in the same column indicate a deletion.

Figure 2.19 also shows the intuition of how to compute this alignment path. The computation proceeds in two steps. In the first step, we augment the minimum edit distance algorithm to store backpointers in each cell. The backpointer from a cell points to the previous cell (or cells) that we came from in entering the current cell. We’ve shown a schematic of these backpointers in Fig. 2.19. Some cells have multiple backpointers because the minimum extension could have come from multiple previous cells. In the second step, we perform a backtrace. In a backtrace, we start

from the last cell (at the final row and column), and follow the pointers back through the dynamic programming matrix. Each complete path between the final cell and the initial cell is a minimum distance alignment. Exercise 2.7 asks you to modify the minimum edit distance algorithm to store the pointers and compute the backtrace to output an alignment.

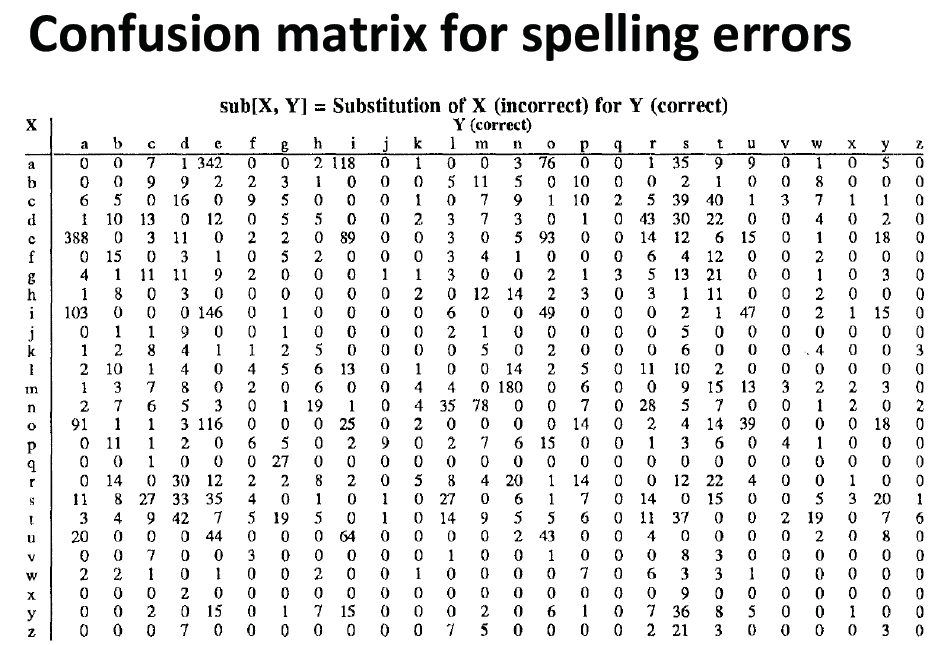


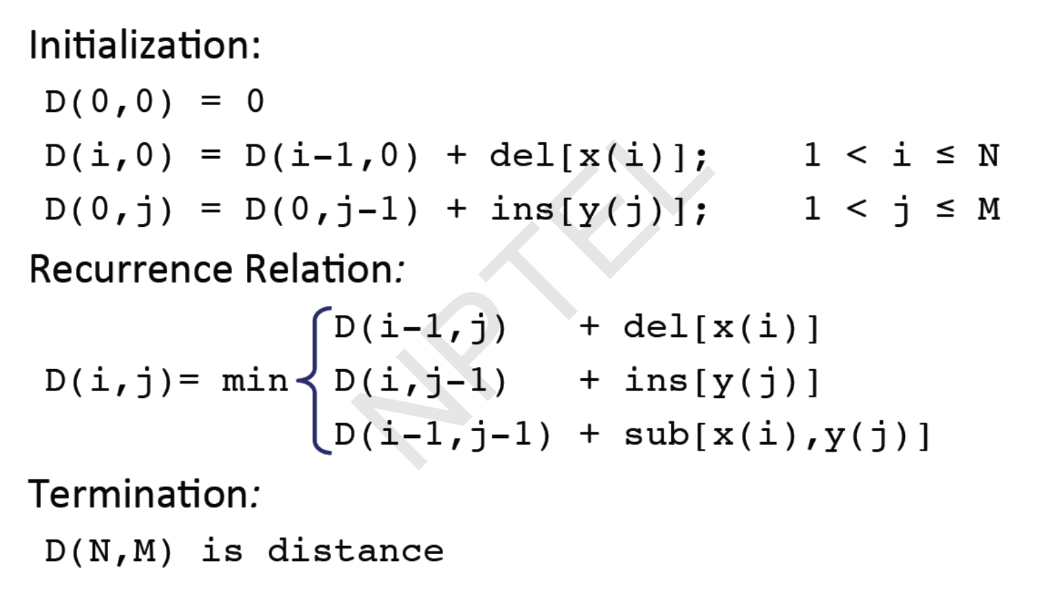
While we worked our example with simple Levenshtein distance, the algorithm in Fig. 2.17 allows arbitrary weights on the operations. For spelling correction, for example, substitutions are more likely to happen between letters that are next to

each other on the keyboard. The Viterbi algorithm is a probabilistic extension of minimum edit distance. Instead of computing the “minimum edit distance” between two strings, Viterbi computes the “maximum probability alignment” of one string with another.

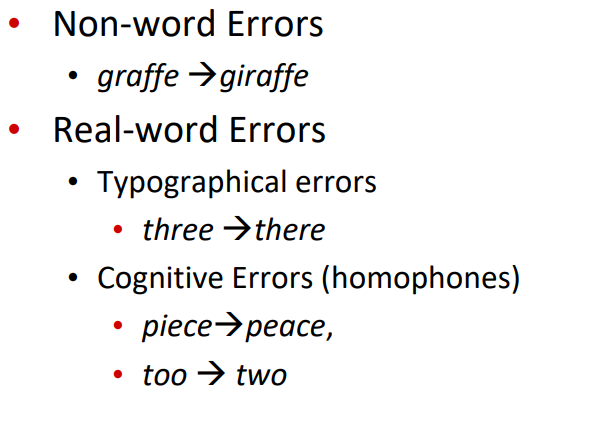
**Weighted Edit Distance:**

In Figure 2.17 the algorithm calculates edit distance in general sense in which we can calculate edit distance with the different weights for insertion, deletion, and replacement. In basic edit distance we have 1 as weights to all kind of operations.

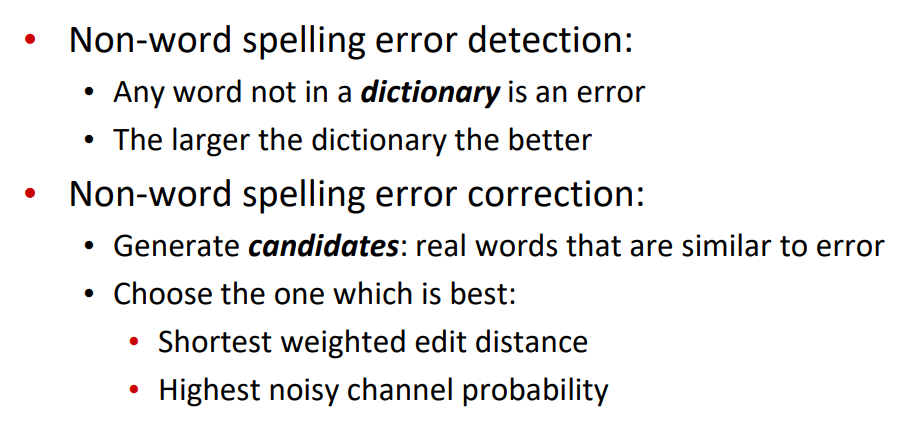




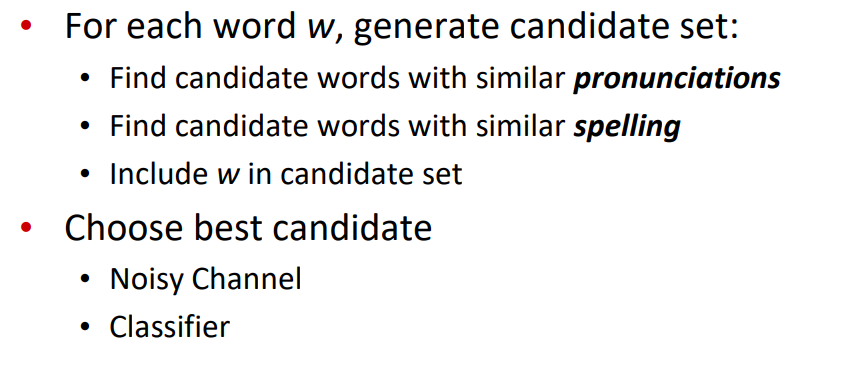
**Types of spelling Errors:**

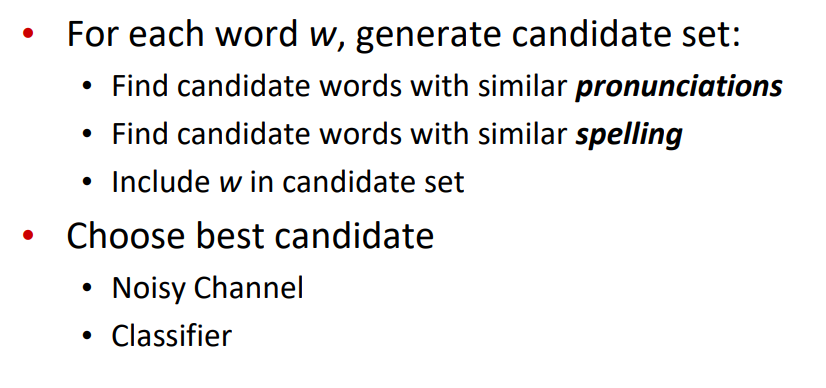


**Non-word Spelling Errors:**



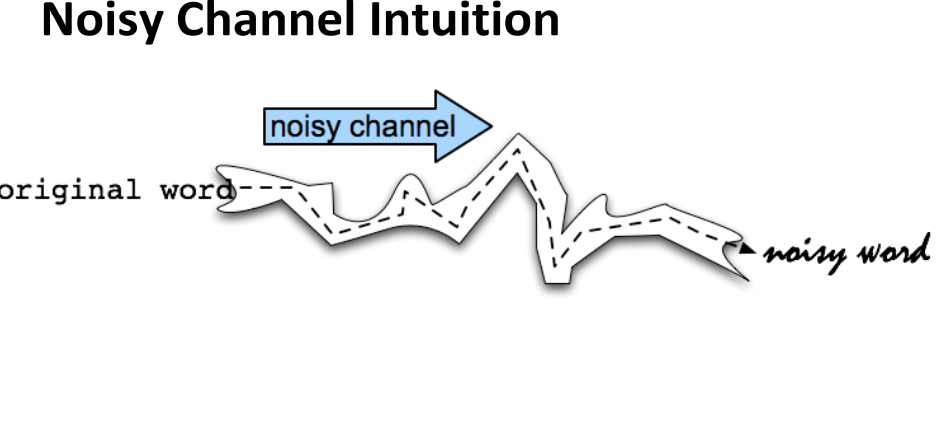
**Real-world spelling errors:**



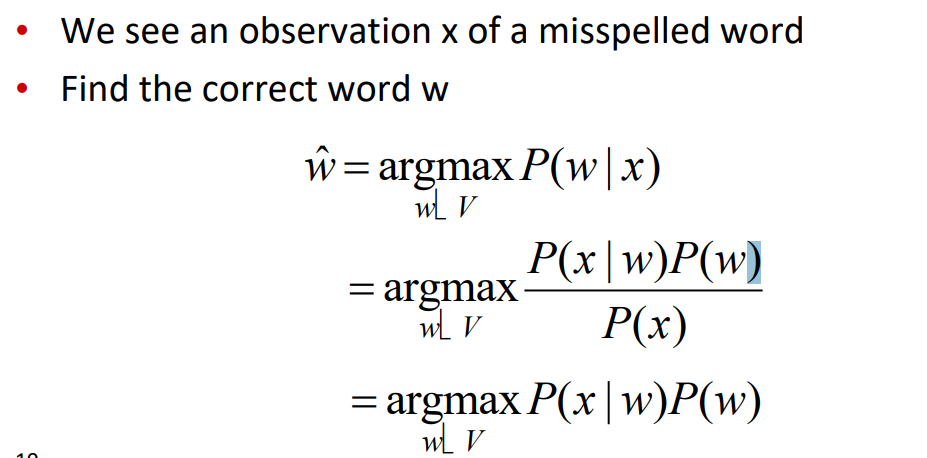


**Spelling Correction in Noisy-Channel:**

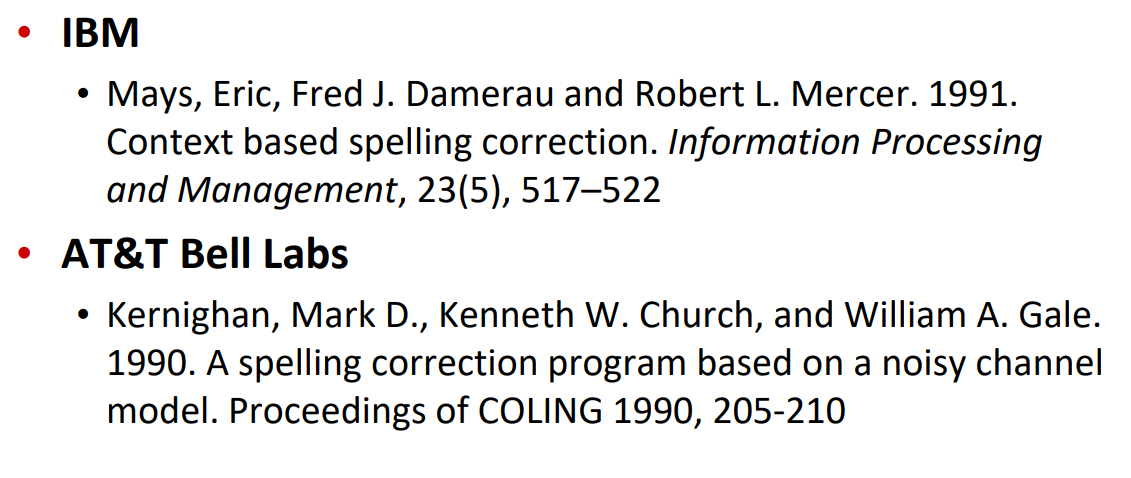
The communication between source and destination may not be ideat, rather most of the time it is noisy, i.e. between communication some sort of noise might be added to the source of the information or words. Here noise may be considered as misspelled word, i.e., one or more wrong letter is inserted in the correct word or some letters are changed.



**Noisy Channel:**



**History of Noisy Channel for spelling proposed around 1990:**

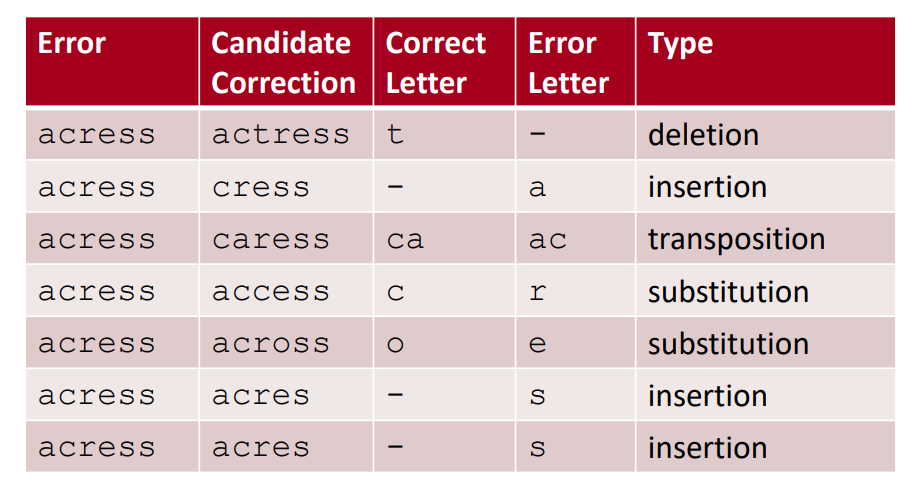


**Non-word spelling error example:**

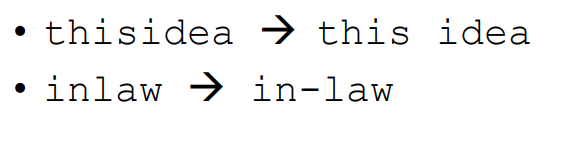
Suppose, we have to correct the spelling error of word “acress”. We have to consider certains points like words with similar spelling i.e. error with small edit distance and small edit distance error of similar pronounce words. The minimal edit distance involves following operations:

1. Insertion
2. Deletion
3. Substitution
4. Transposition of two adjacent letters

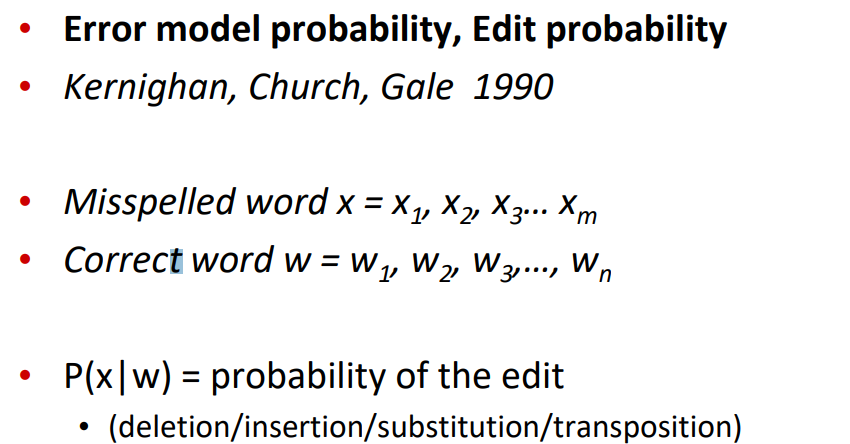
Lets find words within 1 of acress



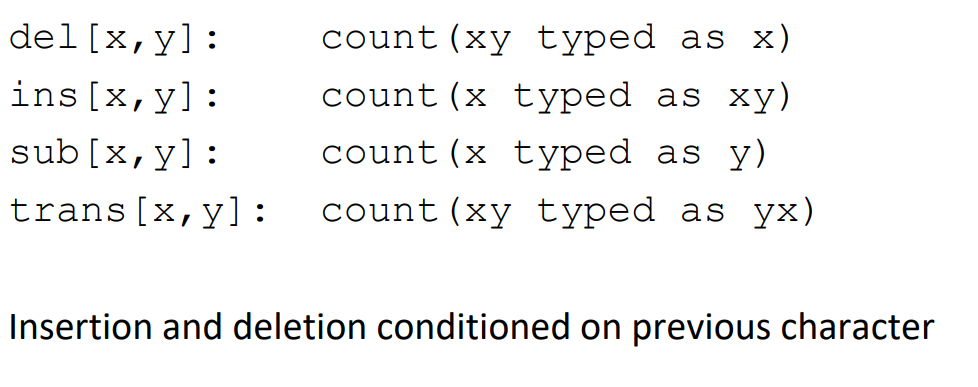
In the previous example we can see that 80% of the errors are within 1 edit distance and alsmost all edit distances are within 2. Also we should allow insertion of space and hyphane

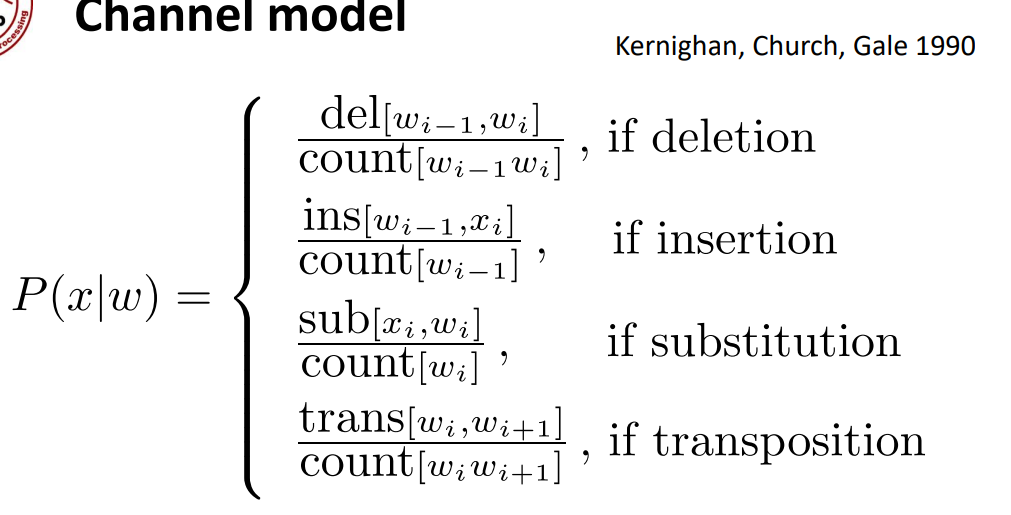


**Channel model probability:**

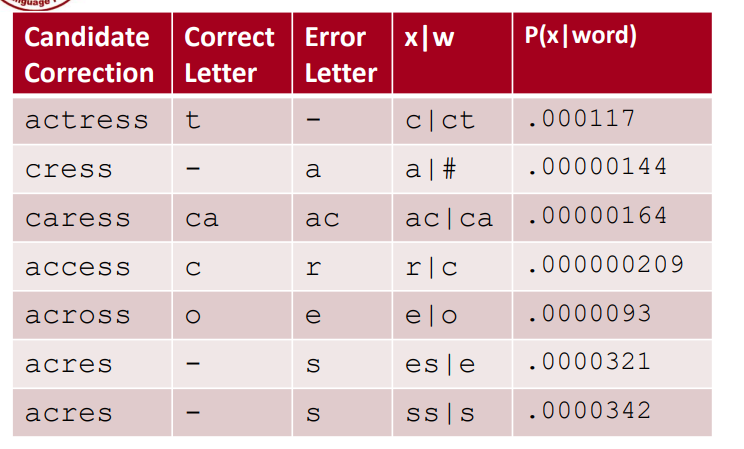


**Computing error probability: confusion matrix**

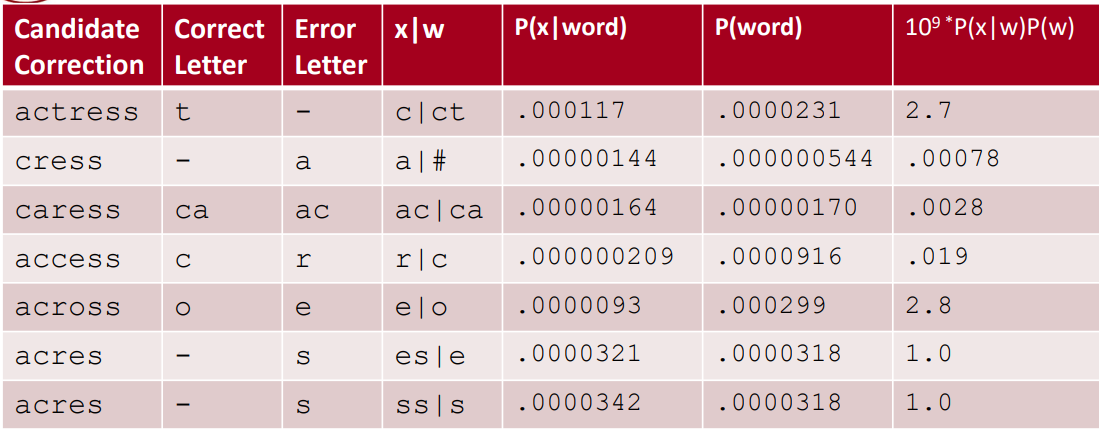




**Channel model for acress**



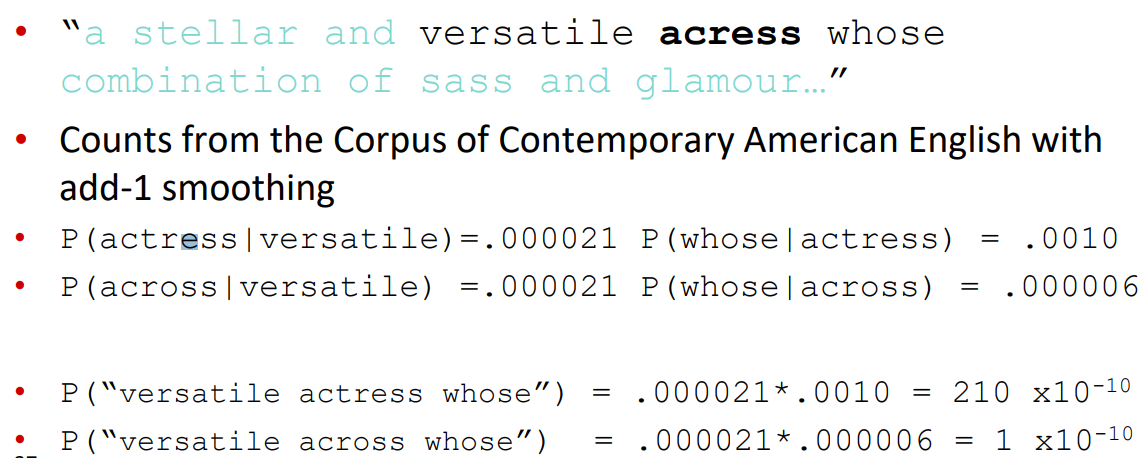
**Noisy channel probability for acress**



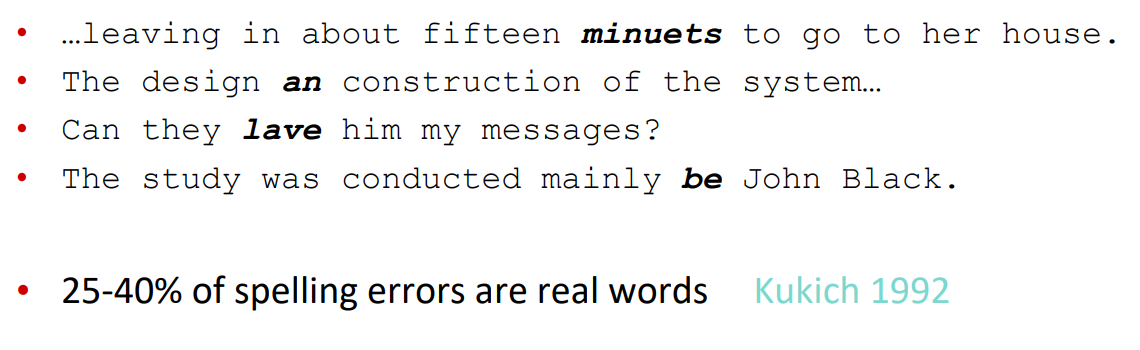
**Noisy channel probability for acress**



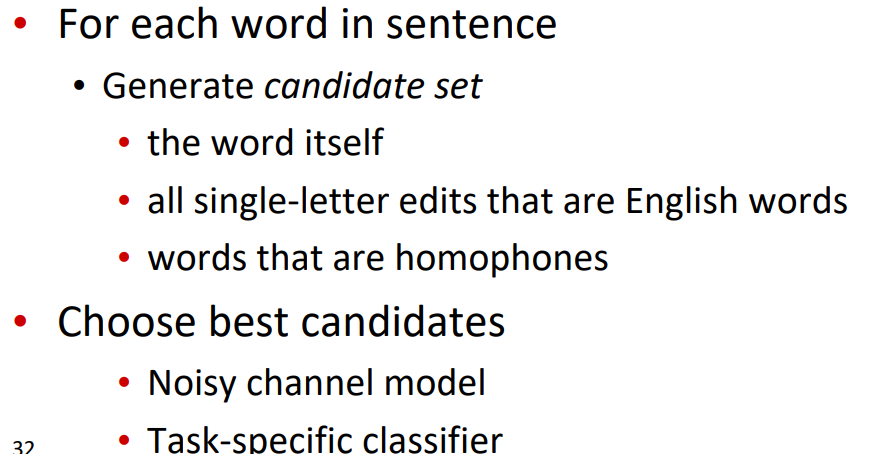
**Using a bigram language model**



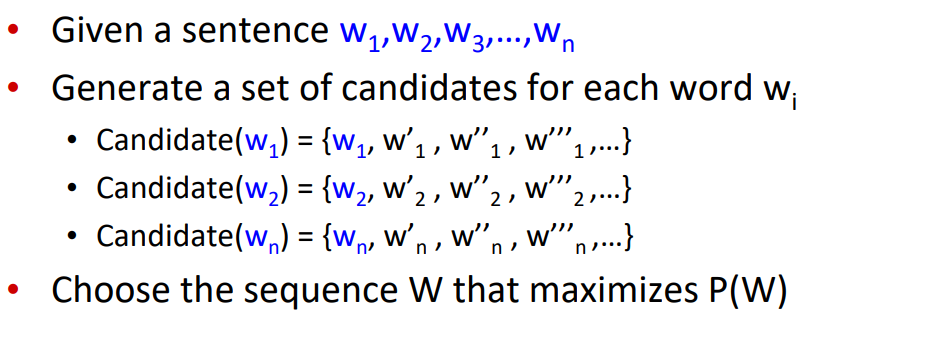
**Real-word spelling errors:**

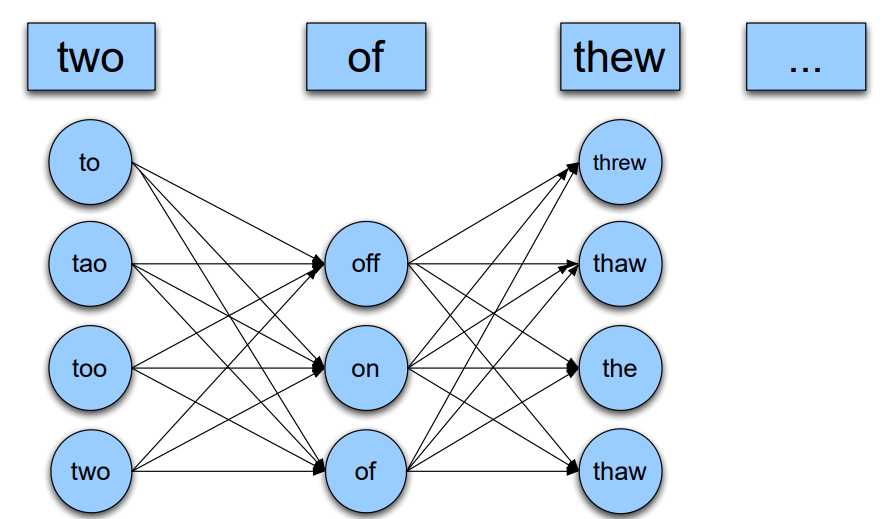


**Solving real-world spelling errors:**

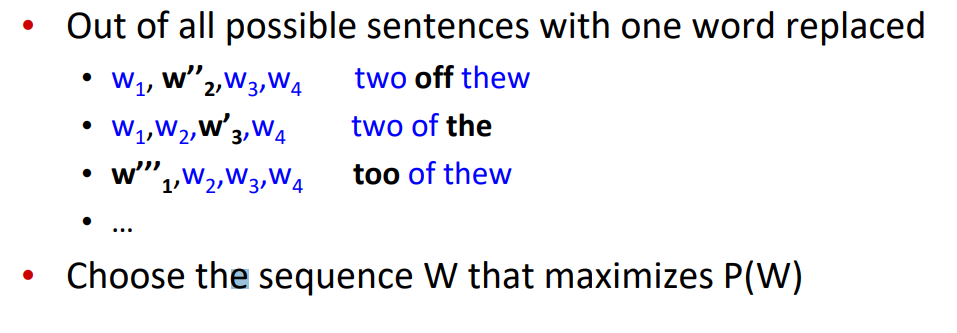


**Noisy channel for real-word spell correction:**

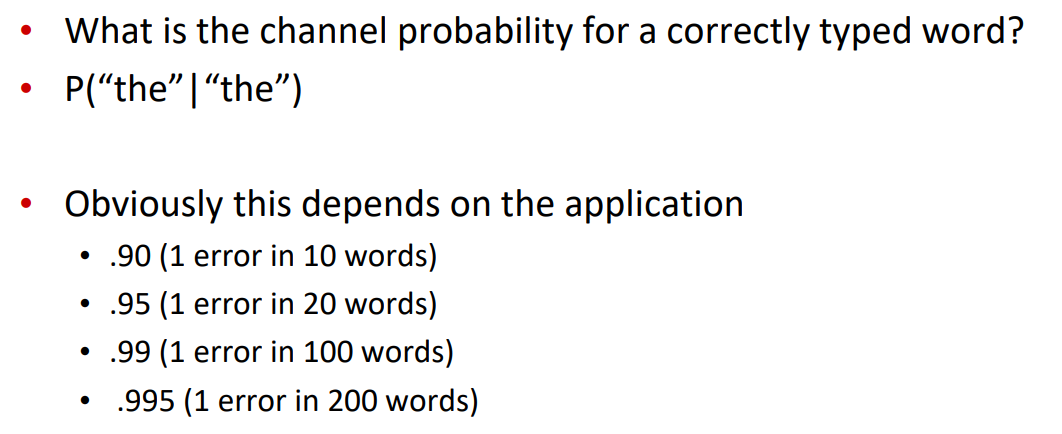




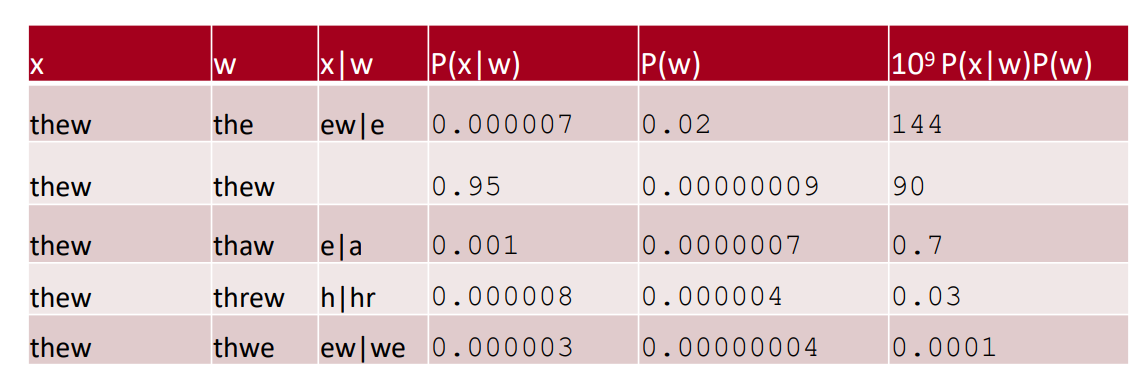
**Simplification: One error per sentence:**



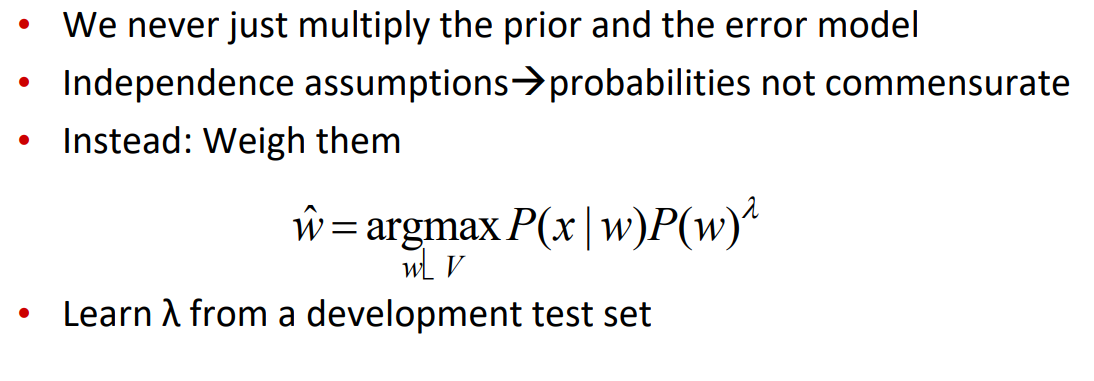
**Probability of no error:**



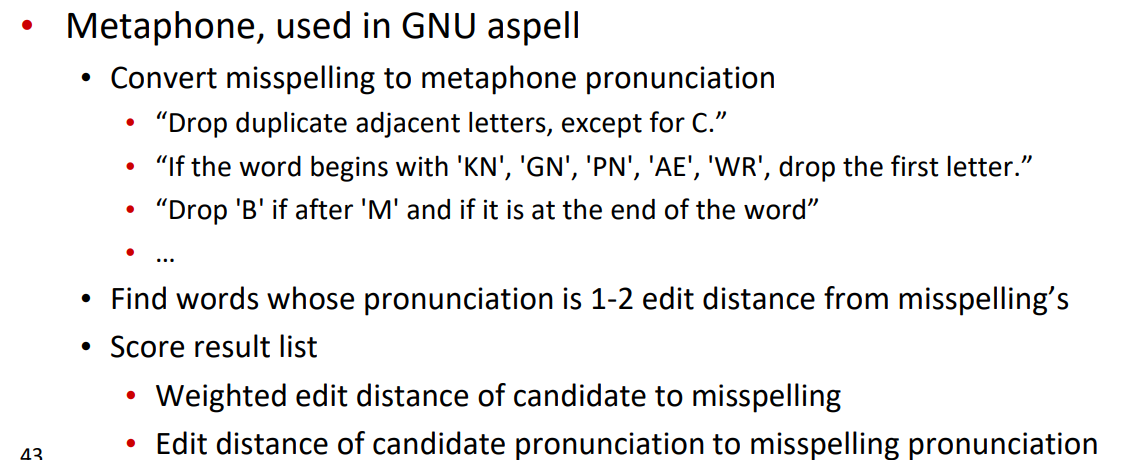
**Peter Norvig’s “thew” example:**



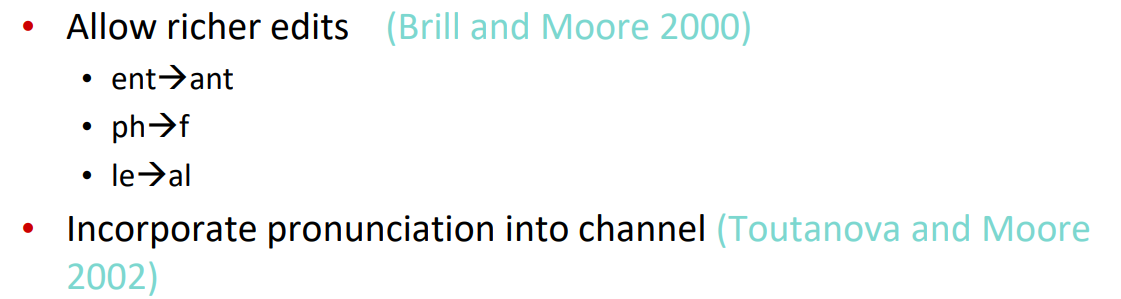
**State of the art noisy channel:**



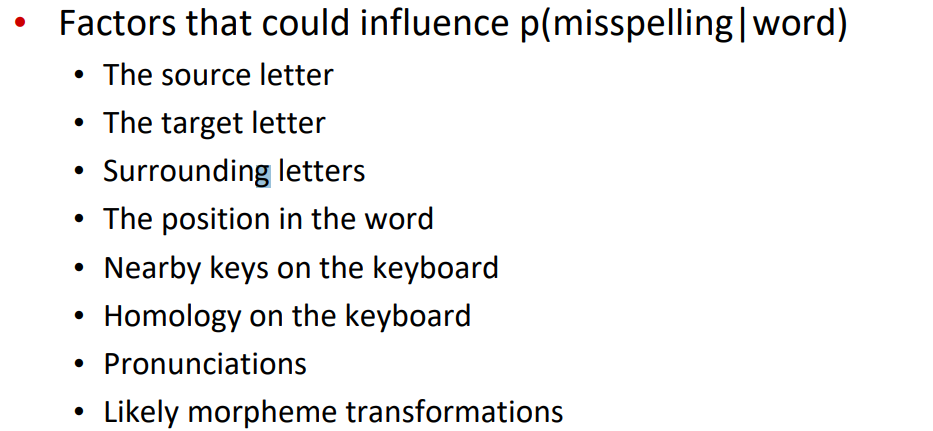
**Phonetic error model:**



**Improvements to channel model:**



**Channel model:**



**N-gram Language Model:**

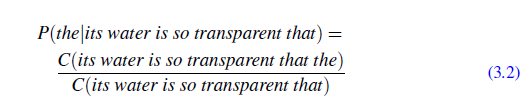
Models that assign probabilities to sequences of words are called language models or LMs. In this section we introduce the simplest model that assigns probabilities to sentences and sequences of words, the n-gram. An n-gram is a sequence of n words: a 2-gram (which we’ll call bigram) is a two-word sequence of words like “please turn”, “turn your”, or ”your homework”, and a 3-gram (a trigram) is a three-word sequence of words like “please turn your”, or “turn your homework”. We’ll see how to use n-gram models to estimate the probability of the last word of an n-gram given the previous words, and also to assign probabilities to entire sequences. In a bit of terminological ambiguity, we usually drop the word “model”,and use the term n-gram (and bigram, etc.) to mean either the word sequence itself or the predictive model that assigns it a probability.

**N-Grams:**

Let’s begin with the task of computing P(wjh), the probability of a word w given some history h. Suppose the history h is “its water is so transparent that” and we want to know the probability that the next word is the:



One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see its water is so transparent that, and count the number of times this is followed by the. This would be answering the question “Out of the times we saw the history h, how many times was it followed by the word w”, as follows:



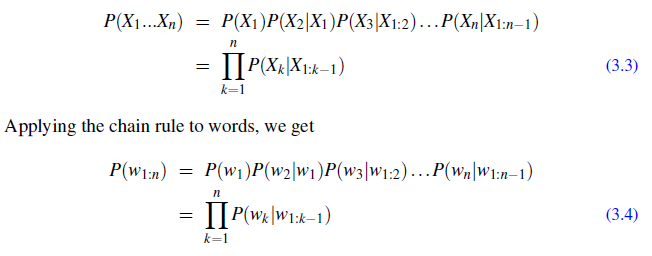
With a large enough corpus, such as the web, we can compute these counts and estimate the probability from Eq. 3.2. You should pause now, go to the web, and compute this estimate for yourself.

While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn’t big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won’t always be able to count entire sentences. Even simple extensions of the example sentence may have counts of zero on the web (such as “Walden Pond’s water is so transparent that the”; well, used to have counts of zero).

Similarly, if we wanted to know the joint probability of an entire sequence of words like its water is so transparent, we could do it by asking “out of all possible sequences of five words, how many of them are its water is so transparent?” We would have to get the count of its water is so transparent and divide by the sum of the counts of all possible five word sequences. That seems rather a lot to estimate!

For this reason, we’ll need to introduce more clever ways of estimating the probability of a word w given a history h, or the probability of an entire word sequence W. Let’s start with a little formalizing of notation. To represent the probability of a particular random variable Xi taking on the value “the”, or P(Xi =“the”), we will use the simplification P(the). We’ll represent a sequence of n words either as w1 : : :wn or w1:n (so the expression w1:n􀀀1 means the string w1;w2; :::;wn􀀀1). For the joint probability of each word in a sequence having a particular value P(X1 = w1;X2 = w2;X3 = w3; :::;Xn = wn) we’ll use P(w1;w2; :::;wn).

Now, how can we compute probabilities of entire sequences like P(w1;w2; :::;wn)? One thing we can do is decompose this probability using the chain rule of probability:



The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. Equation 3.4 suggests that we could estimate the joint probability of an entire sequence of words by multiplying together a number of conditional probabilities. But using the chain rule doesn’t really seem to help us! We don’t know any way to compute the exact probability of a word given a long sequence of preceding words, P(wnjw1:n􀀀1). As we said above, we can’t just estimate by counting the number of times every word

occurs following every long string, because language is creative and any particular context might have never occurred before!

The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words.

The bigram model, for example, approximates the probability of a word given all the previous words P(wnjw1:n􀀀1) by using only the conditional probability of the preceding word P(wnjwn􀀀1). In other words, instead of computing the probability



When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

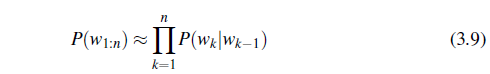


The assumption that the probability of a word depends only on the previous word is called a Markov assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the n-gram (which looks n􀀀1 words into the past).

Let’s see a general equation for this n-gram approximation to the conditional probability of the next word in a sequence. We’ll use N here to mean the n-gram size, so N = 2 means bigrams and N = 3 means trigrams. Then we approximate the probability of a word given its entire context as follows:

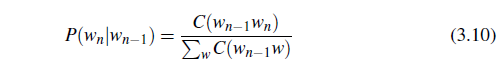


Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by substituting Eq. 3.7 into Eq. 3.4:



How do we estimate these bigram or n-gram probabilities? An intuitive way to estimate probabilities is called maximum likelihood estimation or MLE. We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1.

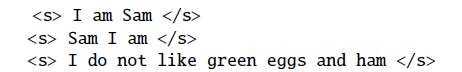
For example, to compute a particular bigram probability of a word wn given a previous word wn􀀀1, we’ll compute the count of the bigram C(wn􀀀1wn) and normalize by the sum of all the bigrams that share the same first word wn􀀀1:

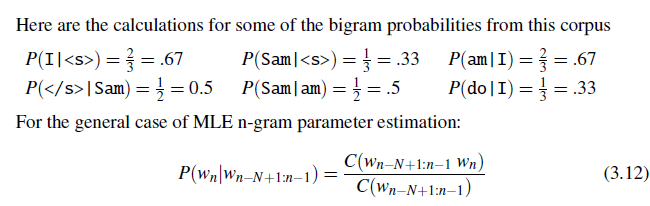


We can simplify this equation, since the sum of all bigram counts that start with a given word wn􀀀1 must be equal to the unigram count for that word wn􀀀1 (the reader should take a moment to be convinced of this):



Let’s work through an example using a mini-corpus of three sentences. We’ll first need to augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word. We’ll also need a special end-symbol. </s>.





Equation 3.12 (like Eq. 3.11) estimates the n-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a relative frequency. We said above that this use of relative frequencies as a way to estimate probabilities is an example of maximum likelihood estimation or MLE. In MLE, the resulting parameter set maximizes the likelihood

of the training set T given the model M (i.e., P(TjM)). For example, suppose the word Chinese occurs 400 times in a corpus of a million words like the Brown corpus. What is the probability that a random word selected from some other text of, say, a million words will be the word Chinese? The MLE of its probability is 400 1000000 or :0004. Now :0004 is not the best possible estimate of the probability of Chinese occurring in all situations; it might turn out that in some other corpus or context Chinese is a very unlikely word. But it is the probability that makes it most likely that Chinese will occur 400 times in a million-word corpus. We present ways to modify the MLE estimates slightly to get better probability estimates in Section 3.5.

Let’s move on to some examples from a slightly larger corpus than our 14-word example above. We’ll use data from the now-defunct Berkeley Restaurant Project, a dialogue system from the last century that answered questions about a database of restaurants in Berkeley, California (Jurafsky et al., 1994). Here are some textnormalized sample user queries (a sample of 9332 sentences is on the website):

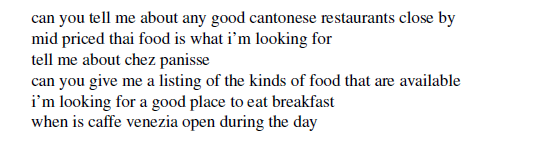
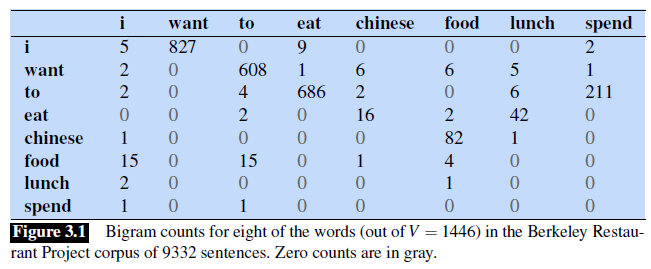
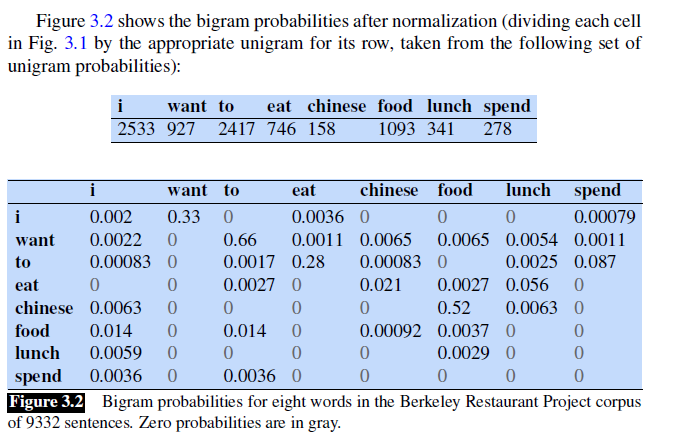
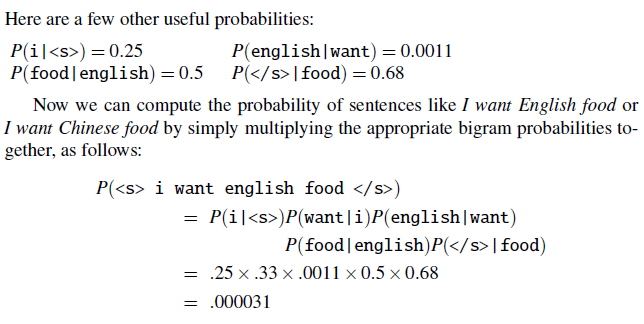


Figure 3.1 shows the bigram counts from a piece of a bigram grammar from the Berkeley Restaurant Project. Note that the majority of the values are zero. In fact, we have chosen the sample words to cohere with each other; a matrix selected from a random set of eight words would be even more sparse.







We leave it as Exercise 3.2 to compute the probability of i want chinese food. What kinds of linguistic phenomena are captured in these bigram statistics? Some of the bigram probabilities above encode some facts that we think of as strictly syntactic in nature, like the fact that what comes after eat is usually a noun or an adjective, or that what comes after to is usually a verb. Others might be a fact about the personal assistant task, like the high probability of sentences beginning with the words I. And some might even be cultural rather than linguistic, like the higher probability that people are looking for Chinese versus English food.

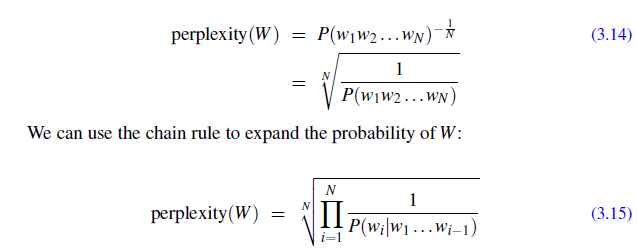
**Evaluating Language Models**

The best way to evaluate the performance of a language model is to embed it in an application and measure how much the application improves. Such end-to-end **extrinsic evaluation** is called extrinsic evaluation. Extrinsic evaluation is the only way to evaluation know if a particular improvement in a component is really going to help the task at hand.

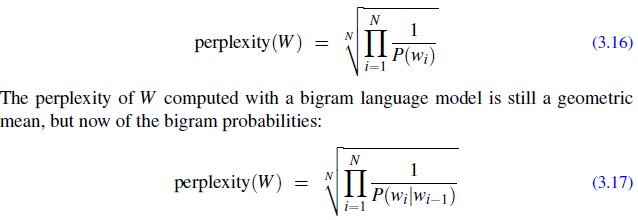
Unfortunately, running big NLP systems end-to-end is often very expensive. Instead, it would be nice to have a metric that can be used to quickly evaluate potential improvements in a language model. An **intrinsic evaluation** metric is one that measures the quality of a model independent of any application.

Perplexity

In practice we don’t use raw probability as our metric for evaluating language models, but a variant called perplexity. The perplexity (sometimes called PPL for short) of a language model on a test set is the inverse probability of the test set, normalized by the number of words. For a test setW = w1w2 : : :wN,:



The perplexity of a test setW depends on which language model we use. Here’s the perplexity ofW with a unigram language model (just the geometric mean of the unigram probabilities):



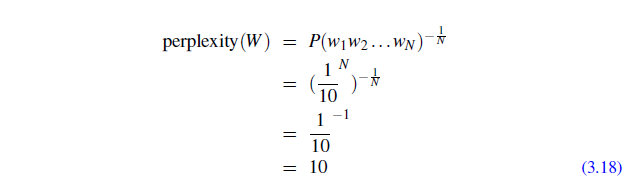
Note that because of the inverse in Eq. 3.15, the higher the conditional probability of the word sequence, the lower the perplexity. Thus, minimizing perplexity is equivalent to maximizing the test set probability according to the language model. What we generally use for word sequence in Eq. 3.15 or Eq. 3.17 is the entire sequence of words in some test set. Since this sequence will cross many sentence boundaries, we need to include the begin- and end-sentence markers <s> and </s>

in the probability computation. We also need to include the end-of-sentence marker </s> (but not the beginning-of-sentence marker <s>) in the total count of word tokens N.

There is another way to think about perplexity: as the weighted average branching factor of a language. The branching factor of a language is the number of possible next words that can follow any word. Consider the task of recognizing the digits in English (zero, one, two,..., nine), given that (both in some training set and in some test set) each of the 10 digits occurs with equal probability P= 1

10 . The perplexity of this mini-language is in fact 10. To see that, imagine a test string of digits of length N, and assume that in the training set all the digits occurred with equal probability.

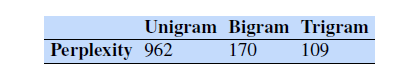
By Eq. 3.15, the perplexity will be



But suppose that the number zero is really frequent and occurs far more often than other numbers. Let’s say that 0 occur 91 times in the training set, and each of the other digits occurred 1 time each. Now we see the following test set: 0 0 0 0 0 3 0 0 0 0. We should expect the perplexity of this test set to be lower since most of the time the next number will be zero, which is very predictable, i.e. has a high probability. Thus, although the branching factor is still 10, the perplexity or weighted branching

factor is smaller. We leave this exact calculation as exercise 3.12.

We mentioned above that perplexity is a function of both the text and the language model: given a textW, different language models will have different perplexities. Because of this, perplexity can be used to compare different n-gram models. Let’s look at an example, in which we trained unigram, bigram, and trigram grammars on 38 million words (including start-of-sentence tokens) from the Wall Street Journal, using a 19,979 word vocabulary. We then computed the perplexity of each of these models on a test set of 1.5 million words, using Eq. 3.16 for unigrams, Eq. 3.17 for bigrams, and the corresponding equation for trigrams. The table below shows the perplexity of a 1.5 million word WSJ test set according to each of these grammars.



As we see above, the more information the n-gram gives us about the word sequence, the higher the probability the n-gram will assign to the string. A trigram model is less surprised than a unigram model because it has a better idea of what words might come next, and so it assigns them a higher probability. And the higher the probability, the lower the perplexity (since as Eq. 3.15 showed, perplexity is related inversely to the likelihood of the test sequence according to the model). So a

lower perplexity can tell us that a language model is a better predictor of the words in the test set.

**Smoothing:**

What do we do with words that are in our vocabulary (they are not unknown words) but appear in a test set in an unseen context (for example they appear after a word they never appeared after in training)? To keep a language model from assigning zero probability to these unseen events, we’ll have to shave off a bit of probability mass from some more frequent events and give it to the events we’ve never seen.This modification is called smoothing or discounting. In this section and the fol

lowing ones we’ll introduce a variety of ways to do smoothing: Laplace (add-one) smoothing, add-k smoothing, stupid backoff, and Kneser-Ney smoothing

**Laplace Smoothing**

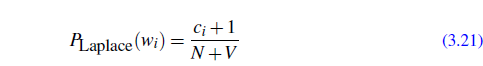
The simplest way to do smoothing is to add one to all the n-gram counts, before we normalize them into probabilities. All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on. This algorithm is called Laplace smoothing. Laplace smoothing does not perform well enough to be used in modern n-gram models, but it usefully introduces many of the concepts that we

see in other smoothing algorithms, gives a useful baseline, and is also a practical smoothing algorithm for other tasks like text classification

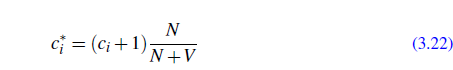
Let’s start with the application of Laplace smoothing to unigram probabilities. Recall that the unsmoothed maximum likelihood estimate of the unigram probability of the word wi is its count ci normalized by the total number of word tokens N:



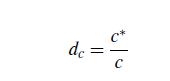
Laplace smoothing merely adds one to each count (hence its alternate name add one smoothing). Since there are V words in the vocabulary and each one was incremented,we also need to adjust the denominator to take into account the extra V observations. (What happens to our P values if we don’t increase the denominator?)



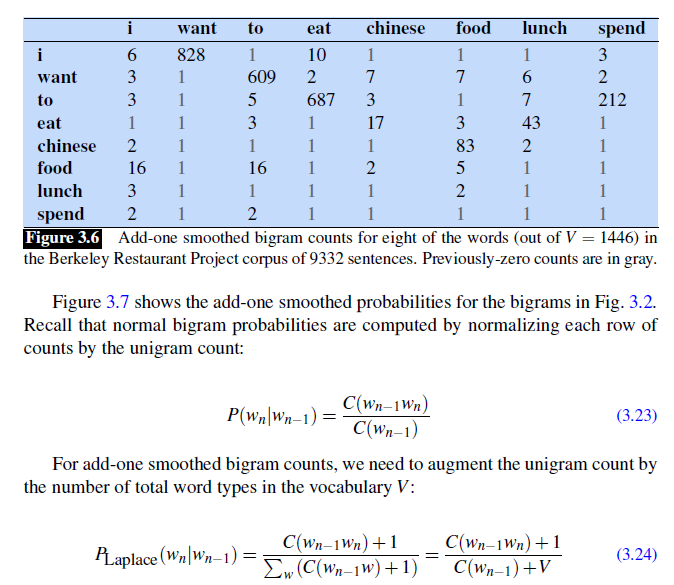
Instead of changing both the numerator and denominator, it is convenient to describe how a smoothing algorithm affects the numerator, by defining an adjusted count c\_. This adjusted count is easier to compare directly with the MLE counts and can be turned into a probability like an MLE count by normalizing by N. To define this count, since we are only changing the numerator in addition to adding 1 we’ll also need to multiply by a normalization factor N N+V :

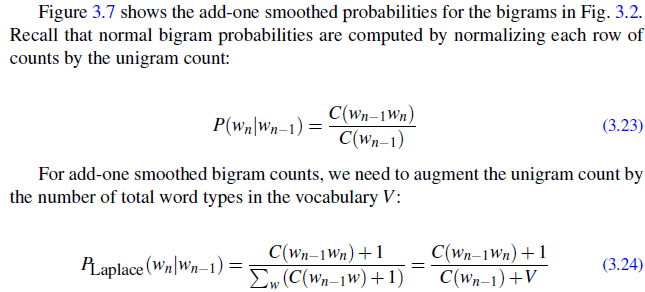


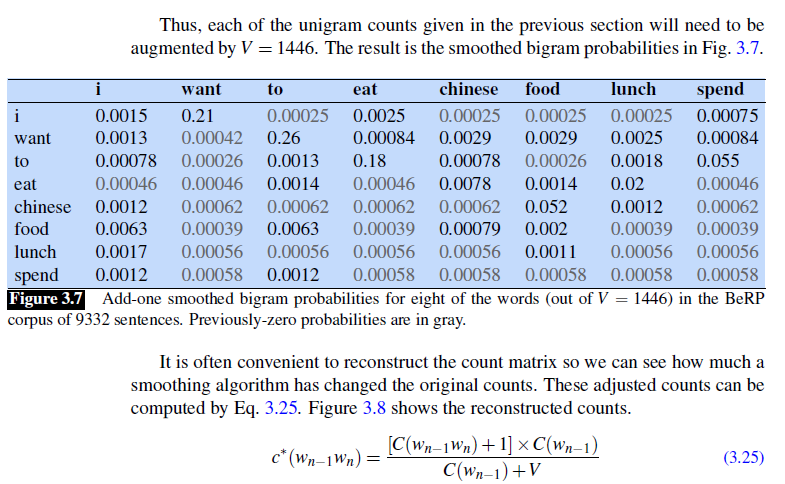
A related way to view smoothing is as discounting (lowering) some non-zero counts in order to get the probability mass that will be assigned to the zero counts. Thus, instead of referring to the discounted counts c\_, we might describe a smoothing algorithm in terms of a relative discount dc, the ratio of the discounted counts to the original counts:



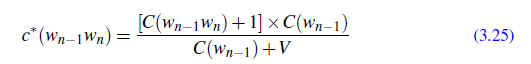
Now that we have the intuition for the unigram case, let’s smooth our Berkeley Restaurant Project bigrams. Figure 3.6 shows the add-one smoothed counts for the bigrams in Fig. 3.1.

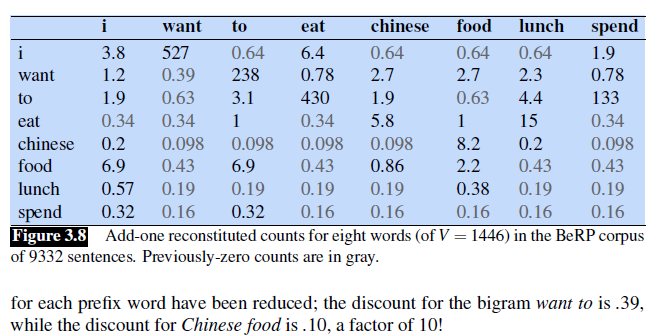






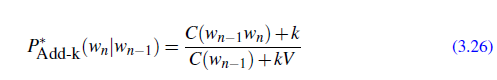
It is often convenient to reconstruct the count matrix so we can see how much a smoothing algorithm has changed the original counts. These adjusted counts can be computed by Eq. 3.25. Figure 3.8 shows the reconstructed counts





**Add-k smoothing**

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events. Instead of adding 1 to each count, we add a fractional count k (.5? .05? .01?). This algorithm is therefore called add-k smoothing



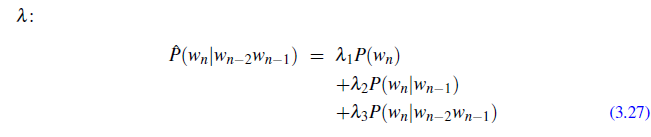
Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a devset. Although add-k is useful for some tasks (including text classification), it turns out that it still doesn’t work well for language modeling, generating counts with poor variances and often inappropriate discounts (Gale and Church, 1994).

**Backoff and Interpolation**

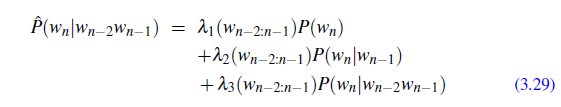
The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on. If we are trying to compute P(wnjwn􀀀2wn􀀀1) but we have no examples of a particular trigram wn􀀀2wn􀀀1wn, we can instead estimate its probability by using the bigram probability P(wnjwn􀀀1). Similarly, if we don’t have counts to compute P(wnjwn􀀀1), we can look to the unigram P(wn).

In other words, sometimes using less context is a good thing, helping to generalize more for contexts that the model hasn’t learned much about. There are two ways to use this n-gram “hierarchy”. In backoff, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only “back off” to a lower-order n-gram if we have zero evidence for a higher-order n-gram. By contrast, in interpolation, we always mix the probability estimates from all the n-gram estimators, weighting and combining the trigram, bigram, and unigram counts.

In simple linear interpolation, we combine different order n-grams by linearly interpolating them. Thus, we estimate the trigram probability P(wnjwn􀀀2wn􀀀1) by mixing together the unigram, bigram, and trigram probabilities, each weighted by a



In a slightly more sophisticated version of linear interpolation, each l weight is computed by conditioning on the context. This way, if we have particularly accurate counts for a particular bigram, we assume that the counts of the trigrams based on this bigram will be more trustworthy, so we can make the ls for those trigrams higher and thus give that trigram more weight in the interpolation. Equation 3.29 shows the equation for interpolation with context-conditioned weights:



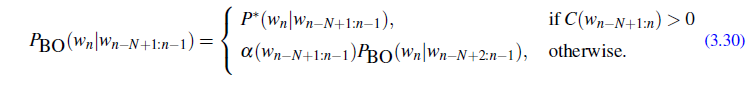
How are these l values set? Both the simple interpolation and conditional interpolation ls are learned from a held-out corpus. A held-out corpus is an additional training corpus, so-called because we hold it out from the training data, that we use to set hyperparameters like these l values. We do so by choosing the l values that maximize the likelihood of the held-out corpus. That is, we fix the n-gram probabilities and then search for the l values that—when plugged into Eq. 3.27—give us the highest probability of the held-out set. There are various ways to find this optimal set of ls. One way is to use the EM algorithm, an iterative learning algorithm that converges on locally optimal ls (Jelinek and Mercer, 1980).

In a backoff n-gram model, if the n-gram we need has zero counts, we approximate it by backing off to the (n-1)-gram. We continue backing off until we reach a history that has some counts.

In order for a backoff model to give a correct probability distribution, we have to discount the higher-order n-grams to save some probability mass for the lower order n-grams. Just as with add-one smoothing, if the higher-order n-grams aren’t discounted and we just used the undiscounted MLE probability, then as soon as we replaced an n-gram which has zero probability with a lower-order n-gram, we would be adding probability mass, and the total probability assigned to all possible strings

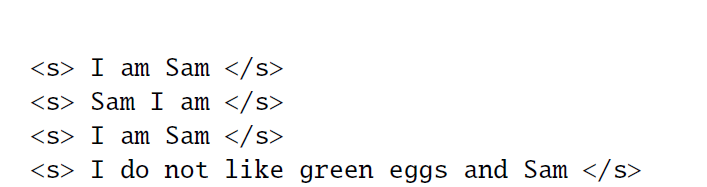
by the language model would be greater than 1! In addition to this explicit discount factor, we’ll need a function a to distribute this probability mass to the lower order n-grams.

This kind of backoff with discounting is also called Katz backoff. In Katz backoff we rely on a discounted probability P\_ if we’ve seen this n-gram before (i.e., if we have non-zero counts). Otherwise, we recursively back off to the Katz probability for the shorter-history (n-1)-gram. The probability for a backoff n-gram PBO is thus computed as follows:



**Important Questions:**

1. Calculate the probability of the sentence i want chinese food. Give two probabilities, one using Fig. 3.2 and the ‘useful probabilities’ just below it on page 36, and another using the add-1 smoothed table in Fig. 3.7. Assume the additional add-1 smoothed probabilities P(i|<s>)=0:19 and P(</s>|food)= 0:40.
2. We are given the following corpus, modified from the one in the chapter:



Using a bigram language model with add-one smoothing, what is P(Sam j am)? Include <s> and </s> in your counts just like any other token.

1. Analyze Laplace smoothing with the help of suitable example.
2. How language models are evaluated explain elaborately?
3. Apply Byte-pair encoding for the following corpus:

“With the Bioethics Unit of the Indian Council of Medical Research (ICMR) placing a consensus policy statement on Controlled Human Infection Studies (CHIS) for comments, India has taken the first step in clearing the deck for such studies to be undertaken here.”

**Interview Questions:**

1. What do you understand by Natural Language Processing?
2. What is n-gram in NLP?
3. What is the corpus in NLP?
4. What is tokenization in NLP?
5. What is perplexity in NLP?

**Project:**

*Build a system that will check spelling of words whether it is correct or wrong from a corpus.*

**Important Research Papers:**

1. Chen, M., Suresh, A. T., Mathews, R., Wong, A., Allauzen, C., Beaufays, F., & Riley, M. (2019). Federated learning of n-gram language models. arXiv preprint arXiv:1910.03432.
2. Wang, S., Chollak, D., Movshovitz-Attias, D., & Tan, L. (2016, August). Bugram: bug detection with n-gram language models. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (pp. 708-719).
3. Etoori, P., Chinnakotla, M., & Mamidi, R. (2018, July). Automatic spelling correction for resource-scarce languages using deep learning. In Proceedings of ACL 2018, Student Research Workshop (pp. 146-152).
4. Guo, J., Sainath, T. N., & Weiss, R. J. (2019, May). A spelling correction model for end-to-end speech recognition. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5651-5655). IEEE.

**Unit 2**

**Advanced Smoothing**

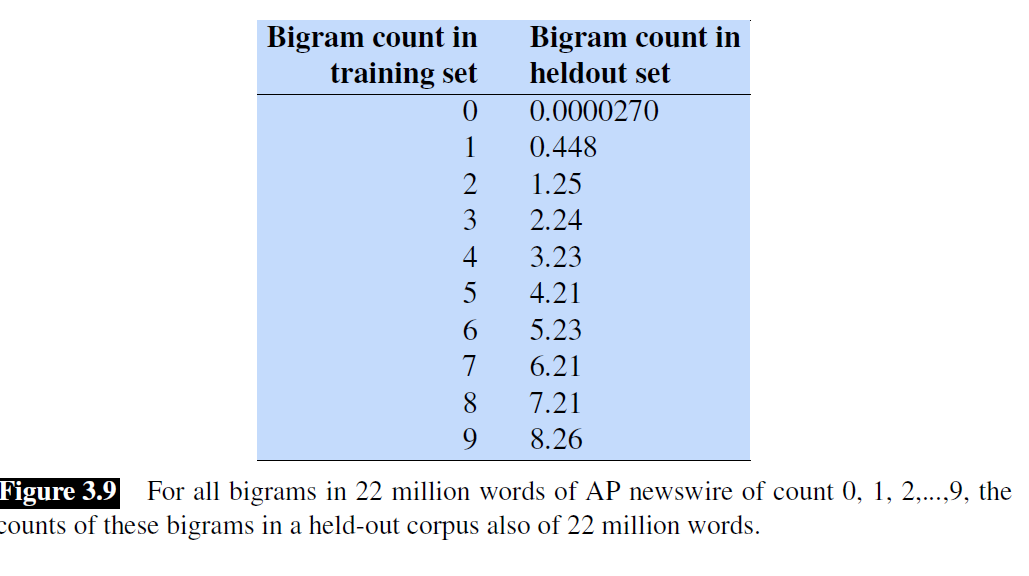
**Kneser-Ney Smoothing**

A popular advanced n-gram smoothing method is the interpolated Kneser-Ney algorithm. (Kneser and Ney 1995, Chen and Goodman 1998).

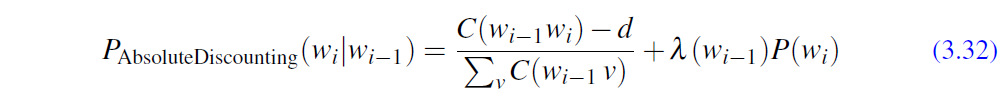
**Absolute Discounting**

Kneser-Ney has its roots in a method called absolute discounting. Recall that discounting of the counts for frequent n-grams is necessary to save some probability mass for the smoothing algorithm to distribute to the unseen n-grams.

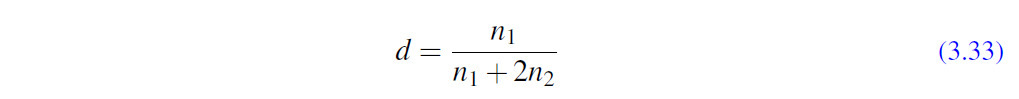
To see this, we can use a clever idea from Church and Gale (1991). Consider an n-gram that has count 4. We need to discount this count by some amount. But how much should we discount it? Church and Gale’s clever idea was to look at a held-out corpus and just see what the count is for all those bigrams that had count 4 in the training set. They computed a bigram grammar from 22 million words of AP newswire and then checked the counts of each of these bigrams in another 22 million words. On average, a bigram that occurred 4 times in the first 22 million words occurred 3.23 times in the next 22 million words. Fig. 3.9 from Church and Gale (1991) shows these counts for bigrams with c from 0 to 9.



Notice in Fig. 3.9 that except for the held-out counts for 0 and 1, all the other bigram counts in the held-out set could be estimated pretty well by just subtracting 0.75 from the count in the training set! Absolute discounting formalizes this intuition by subtracting a fixed (absolute) discount d from each count. The intuition is that since we have good estimates already for the very high counts, a small discount d won’t affect them much. It will mainly modify the smaller counts, for which we don’t necessarily trust the estimate anyway, and Fig. 3.9 suggests that in practice this discount is actually a good one for bigrams with counts 2 through 9. The equation for interpolated absolute discounting applied to bigrams:



The first term is the discounted bigram, with . and the second term is the unigram with an interpolation weight . By inspection of Fig. 3.9, it looks like just setting all the d values to .75 would work very well, or perhaps keeping a separate second discount value of 0.5 for the bigrams with counts of 1. There are principled methods for setting d; for example, Ney et al. (1994) set d as a function of and , the number of unigrams that have a count of 1 and a count of 2, respectively:



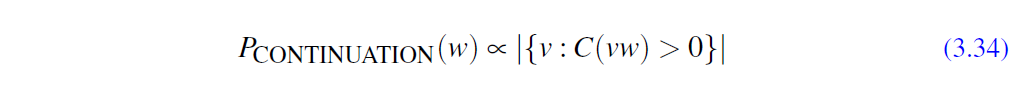
**Kneser-Ney Discounting**

Kneser-Ney discounting (Kneser and Ney, 1995) augments absolute discounting with a more sophisticated way to handle the lower-order unigram distribution. Consider the job of predicting the next word in this sentence, assuming we are interpolating a bigram and a unigram model.



The word glasses seems much more likely to follow here than, say, the word Kong, so we’d like our unigram model to prefer glasses. But in fact it’s Kong that is more common, since Hong Kong is a very frequent word. A standard unigram model will assign Kong a higher probability than glasses. We would like to capture the intuition that although Kong is frequent, it is mainly only frequent in the phrase Hong Kong, that is, after the word Hong. The word glasses has a much wider distribution.

In other words, instead of P(w), which answers the question “How likely is w?”, we’d like to create a unigram model that we might call PCONTINUATION, which answers the question “How likely is w to appear as a novel continuation?”. How can we estimate this probability of seeing the word w as a novel continuation, in a new unseen context? The Kneser-Ney intuition is to base our estimate of PCONTINUATION on the number of different contexts word w has appeared in, that is, the number of bigram types it completes. Every bigram type was a novel continuation the first time it was seen. We hypothesize that words that have appeared in more contexts in the past are more likely to appear in some new context as well. The number of times a word w appears as a novel continuation can be expressed as:



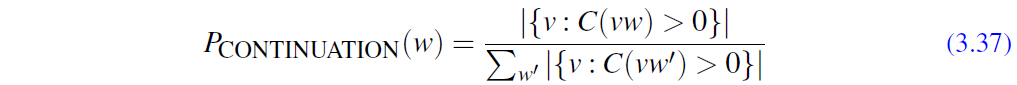
To turn this count into a probability, we normalize by the total number of word bigram types. In summary:



An equivalent formulation based on a different metaphor is to use the number of word types seen to precede w (Eq. 3.34 repeated):

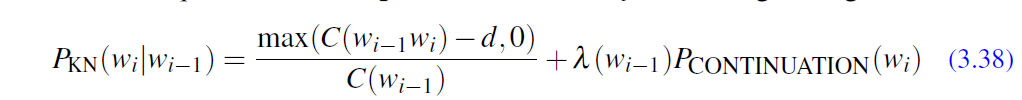


normalized by the number of words preceding all words, as follows:

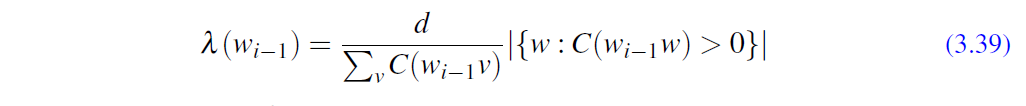


A frequent word (Kong) occurring in only one context (Hong) will have a low continuation probability.

The final equation for Interpolated Kneser-Ney smoothing for bigrams is then:

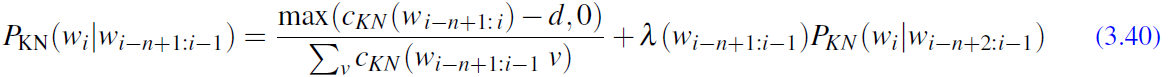


The is a normalizing constant that is used to distribute the probability mass we’ve discounted:

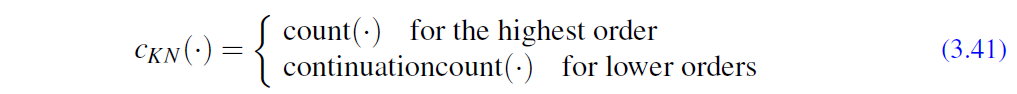


The first term,, is the normalized discount (the discount d,, was introduced in the absolute discounting section above). The second term,, is the number of word types that can follow , or, equivalently, the number of word types that we discounted; in other words, the number of times we applied the normalized discount.

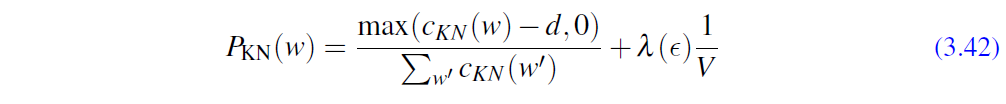
The general recursive formulation is as follows:



where the definition of the count depends on whether we are counting the highest-order n-gram being interpolated (for example trigram if we are interpolating trigram, bigram, and unigram) or one of the lower-order n-grams (bigram or unigram if we are interpolating trigram, bigram, and unigram):



At the termination of the recursion, unigrams are interpolated with the uniform distribution, where the parameter is the empty string:



**Computational Morphology:**

**Morphological analysi**s is a field of linguistics that studies the structure of words. It identifies how a word is produced through the use of morphemes. A **morpheme** is a basic unit of the English language. The morpheme is the smallest element of a word that has grammatical function and meaning. *Free morpheme and bound morpheme are the two types of morphemes. A single free morpheme can become a complete word.*

For instance, a bus, a bicycle, and so forth. A bound morpheme, on the other hand, cannot stand alone and must be joined to a free morpheme to produce a word. ing, un, and other bound morphemes are examples.

**Types of Morphology:**

1. ***Inflectional Morphology:*** modification of a word to express different grammatical categories. Inflectional morphology is the study of processes, including affixation and vowel change, that distinguish word forms in certain grammatical categories. Inflectional morphology consists of at least five categories, provided in the following excerpt from Language Typology and Syntactic Description: Grammatical Categories and the Lexicon. As the text will explain, derivational morphology cannot be so easily categorized because derivation isn’t as predictable as inflection.Examples- cats, men etc.
2. ***Derivational Morphology:*** Is defined as morphology that creates new lexemes, either by changing the syntactic category (part of speech) of a base or by adding substantial, nongrammatical meaning or both. On the one hand, derivation may be distinguished from inflectional morphology, which typically does not change category but rather modifies lexemes to fit into various syntactic contexts; inflection typically expresses distinctions like number, case, tense, aspect, person, among others. On the other hand, derivation may be distinguished from compounding, which also creates new lexemes, but by combining two or more bases rather than by affixation, reduplication, subtraction, or internal modification of various sorts. Although the distinctions are generally useful, in practice applying them is not always easy.

**APPROACHES TO MORPHOLOGY:**

1. **Morpheme Based Morphology :** In these words are analyzed as arrangements of morphemes.Word-based morphology is (usually) a word-and-paradigm approach. The theory takes paradigms as a central notion. Instead of stating rules to combine morphemes into word forms or to generate word forms from stems, word-based morphology states generalizations that hold between the forms of inflectional paradigms.
2. **Lexeme Based Morphology:** Lexeme-based morphology usually takes what it is called an “item-andprocess” approach. Instead of analyzing a word form as a set of morphemes arranged in sequence , aword form is said to be the result of applying rules that alter a word-form or steam in order to produce a new one.
3. ***Word based Morphology :*** Word-based morphology is usually a word-and -paradigm approach.instead of stating rules to combine morphemes into word forms.

**Lemmatization** is the task of determining that two words have the same root, despite their surface differences. The words am, are, and is have the shared lemma be; the words dinner and dinners both have the lemma dinner. Lemmatizing each of these forms to the same lemma will let us find all mentions of words in Polish like Warsaw. The lemmatized form of a sentence like He is reading detective stories would thus be He be read detective story.

How is lemmatization done? The most sophisticated methods for lemmatization involve complete **morphological parsing** of the word. **Morphology** is the study of morpheme the way words are built up from smaller meaning-bearing units called **morphemes**.Two broad classes of morphemes can be distinguished: **stems**—the central morpheme of the word, supplying the main meaning—and **affixes**—adding “additional” meanings of various kinds. So, for example, the word fox consists of one morpheme (the morpheme fox) and the word cats consists of two: the morpheme cat and the morpheme -s. A morphological parser takes a word like cats and parses it into the two morphemes *cat* and s, or parses a Spanish word like *amaren* (‘if in the future they would love’) into the morpheme *amar* ‘to love’, and the morphological features 3PL and *future subjunctive*.

**The Porter Stemmer**

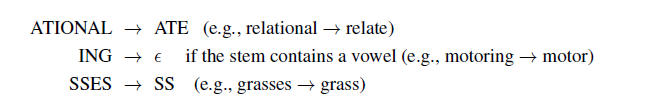
Lemmatization algorithms can be complex. For this reason we sometimes make use of a simpler but cruder method, which mainly consists of chopping off word final affixes. This naive version of morphological analysis is called stemming. For, the Porter stemmer, a widely used stemming algorithm (Porter, 1980),when applied to the following paragraph:

*This was not the map we found in Billy Bones's chest, but an accurate copy, complete in all things-names and heights and soundings-with the single exception of the red crosses and the written notes*.

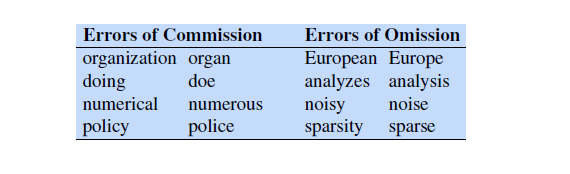
Produces the following output:

*Thi wa not the map we found in Billi Bone s chest but an accur copi complet in all thing name and height and sound with the singl except of the red cross and the written note*

The algorithm is based on series of rewrite rules run in series: the output of each pass is fed as input to the next pass. Here are some sample rules (more details canbe found at <https://tartarus.org/martin/PorterStemmer/>):

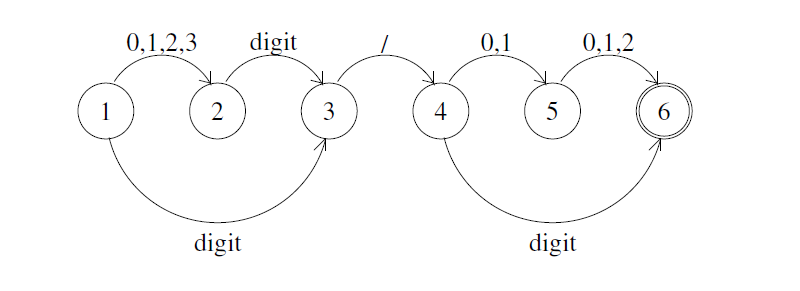


Simple stemmers can be useful in cases where we need to collapse across different variants of the same lemma. Nonetheless, they do tend to commit errors of both over- and under-generalizing, as shown in the table below (Krovetz, 1993):



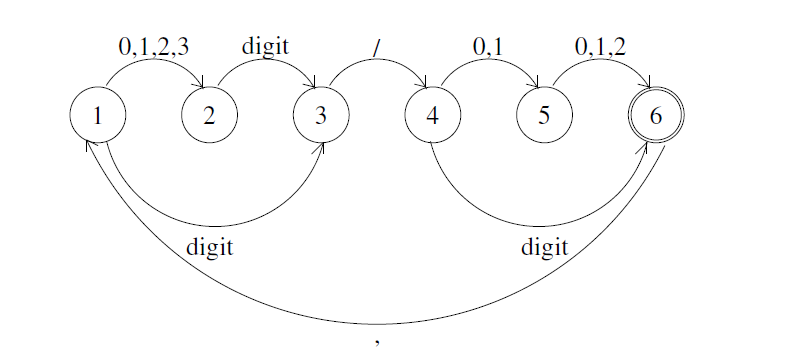
**Finite state automata for recognition**

The approach to spelling rules that I’ll describe involves the use of ﬁnite state transducers (FSTs). Rather than jumping straight into this, I’ll brieﬂy consider the simpler ﬁnite state automata and how they can be used in a simple recogniser. Suppose we want to recognise dates (just day and month pairs) written in the format day/month. The day and the month may be expressed as one or two digits (e.g. 11/2, 1/12 etc). This format corresponds to the following simple FSA, where each character corresponds to one transition:



Accept states are shown with a double circle. This is a non-deterministic FSA: for instance, an input starting with the digit 3 will move the FSA to both state 2 and state 3. This corresponds to a local ambiguity: i.e., one that will be resolved by subsequent context. By convention, there must be no ‘left over’ characters when the system is in the ﬁnal state.

To make this a bit more interesting, suppose we want to recognise a comma-separated list of such dates. The FSA, shown below, now has a cycle and can accept a sequence of indeﬁnite length (note that this is iteration and not full recursion, however).



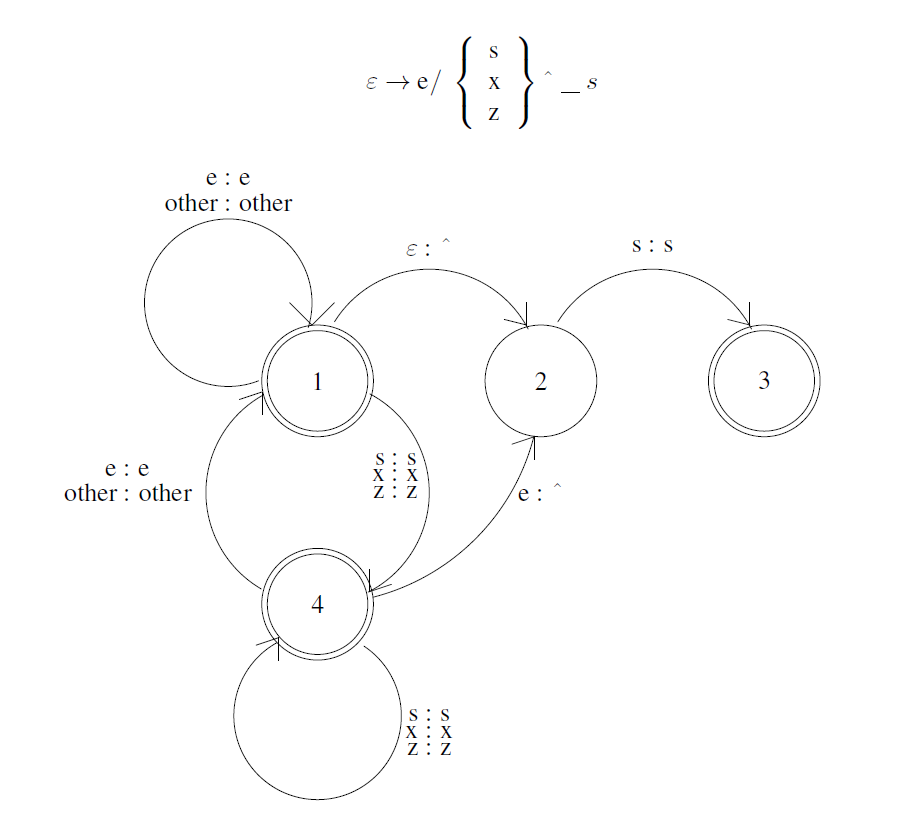
Both these FSAs will accept sequences which are not valid dates, such as 37/00. Conversely, if we use them to generate (random) dates, we will get some invalid output. In general, a system which generates output which is invalid is said to overgenerate. In fact, in many language applications, some amount of overgeneration can be tolerated, especially if we are only concerned with analysis.

**Finite state transducers:**

FSAs can be used to recognise particular patterns, but don’t, by themselves, allow for any analysis of word forms. Hence for morphology, we use ﬁnite state transducers (FSTs) which allow the surface structure to be mapped into the list of morphemes. FSTs are useful for both analysis and generation, since the mapping is bidirectional. This approach is known as two-level morphology.

To illustrate two-level morphology, consider the following FST, which recognises the afﬁx -s allowing for environ-ments corresponding to the e-insertion spelling rule shown in §1.4 and repeated below4

4Actually, I’ve simpliﬁed this slightly so the FST works correctly but the correspondence to the spelling rule is not exact: J&M give a more complex transducer which is an accurate reﬂection of the spelling rule. They also use an explicit terminating character while I prefer to rely on the ‘use all the input’ convention, which results in simpler rules.



transducers map between two representations, so each transition corresponds to a pair of characters. As with the spelling rule, we use the special character ‘ε’ to correspond to the empty character and ‘ˆ’ to correspond to an afﬁx boundary. The abbreviation ‘other : other’ means that any character not mentioned speciﬁcally in the FST maps to itself. As with the FSA example, we assume that the FST only accepts an input if the end of the input corresponds to an accept state (i.e., no ‘left-over’ characters are allowed).

For instance, with this FST, the surface form cakes would start from 1 and go through the transitions/states (c:c) 1,(a:a) 1, (k:k) 1, (e:e) 1, (ε:ˆ) 2, (s:s) 3 (accept, underlying cakeˆs) and also (c:c) 1, (a:a) 1, (k:k) 1, (e:e) 1, (s:s) 4 (accept, underlying cakes). ‘d o g s’ maps to ‘d o g ˆ s’, ‘f o x e s’ maps to ‘f o x ˆ s’ and to ‘f o x e ˆ s’, and ‘b u z z e s’ maps to ‘b u z z ˆ s’ and ‘b u z z e ˆ s’.5 When the transducer is run in analysis mode, this means the system can detect an afﬁx boundary (and hence look up the stem and the afﬁx in the appropriate lexicons). In generation mode, it can construct the correct string. This FST is non-deterministic.

Similar FSTs can be written for the other spelling rules for English (although to do consonant doubling correctly, in-formation about stress and syllable boundaries is required and there are also differences between British and American spelling conventions which complicate matters). Morphology systems are usually implemented so that there is one FST per spelling rule and these operate in parallel.

One issue with this use of FSTs is that they do not allow for any internal structure of the word form. For instance, we can produce a set of FSTs which will result in unionised being mapped into unˆionˆiseˆed, but as we’ve seen, the afﬁxes actually have to be applied in the right order and the bracketing isn’t modelled by the FSTs.

**Introduction to POS Tagging**

Dionysius Thrax of Alexandria (c. 100 B.C.), or perhaps someone else (it was a long time ago), wrote a grammatical sketch of Greek (a “techn¯e”) that summarized the linguistic knowledge of his day. This work is the source of an astonishing proportion of modern linguistic vocabulary, including the words syntax, diphthong, clitic, and parts of speech analogy. Also included are a description of eight parts of speech: noun, verb, pronoun, preposition, adverb, conjunction, participle, and article. Although earlier scholars (including Aristotle as well as the Stoics) had their own lists of parts of speech, it was Thrax’s set of eight that became the basis for descriptions of European languages for the next 2000 years. (All the way to the Schoolhouse Rock educational television shows of our childhood, which had songs about 8 parts of speech, like the late great Bob Dorough’s Conjunction Junction.) The durability of parts of speech through two millennia speaks to their centrality in models of human language.

Parts of speech (also known as POS) and named entities are useful clues to sentence structure and meaning. Knowing whether a word is a noun or a verb tells us about likely neighboring words (nouns in English are preceded by determiners and adjectives, verbs by nouns) and syntactic structure (verbs have dependency links to nouns), making part-of-speech tagging a key aspect of parsing. Knowing if a named entity like Washington is a name of a person, a place, or a university is important to many natural language processing tasks like question answering, stance detection, or information extraction.

**(Mostly) English Word Classes:**

Until now we have been using part-of-speech terms like noun and verb rather freely. In this section we give more complete definitions. While word classes do have semantic tendencies—adjectives, for example, often describe properties and nouns people— parts of speech are defined instead based on their grammatical relationship with neighboring words or the morphological properties about their affixes.

Parts of speech fall into two broad categories: closed class and open class. Closed classes are those with relatively fixed membership, such as prepositions. new prepositions are rarely coined. By contrast, nouns and verbs are open classes—new nouns and verbs like iPhone or to fax are continually being created or borrowed. Closed class words are generally function words like of, it, and, or you, which tend. Closed class words are generally function words like of, it, and, or you, which tend

Nouns are words for people, places, or things, but include others as well. Com-common nouns include concrete terms like cat and mango, abstractions like algorithm. Nouns are words for people, places, or things, but include others as well. Com-common noun mon nouns include concrete terms like cat and mango, abstractions like algorithm. Many languages, including English, divide common nouns into count nouns and mass noun mass nouns. Count nouns can occur in the singular and plural (goat/goats, rela-tionship/relationships) and can be counted (one goat, two goats). Mass nouns are used when something is conceptualized as a homogeneous group. So snow, salt, and proper noun communism are not counted (i.e., \*two snows or \*two communisms). Proper nouns, like Regina, Colorado, and IBM, are names of specific persons or entities.

Verbs refer to actions and processes, including main verbs like draw, provide, and go. English verbs have inflections (non-third-person-singular (eat), third-person-singular (eats), progressive (eating), past participle (eaten)). While many scholars believe that all human languages have the categories of noun and verb, others have argued that some languages, such as Riau Indonesian and Tongan, don’t even make this distinction (Broschart 1997; Evans 2000; Gil 2000) .

Adjectives often describe properties or qualities of nouns, like color (white, black), age (old, young), and value (good, bad), but there are languages without adjectives. In Korean, for example, the words corresponding to English adjectives act as a subclass of verbs, so what is in English an adjective “beautiful” acts in Korean like a verb meaning “to be beautiful”.

Adverbs are a hodge-podge. All the italicized words in this example are adverbs:

Actually, I ran home extremely quickly yesterday

Adverbs generally modify something (often verbs, hence the name “adverb”, but addverbs (home, here, downhill) specify the direction or location of some action; degree adverbs (extremely, very, somewhat) specify the extent of some action, process, or manner property; manner adverbs (slowly, slinkily, delicately) describe the manner of some temporal action or process; and temporal adverbs describe the time that some action or event took place (yesterday, Monday).

Interjections (oh, hey, alas, uh, um) are a smaller open class that also includes greetings (hello, goodbye) and question responses (yes, no, uh-huh).

English adpositions occur before nouns, hence are called prepositions. They can indicate spatial or temporal relations, whether literal (on it, before then, by the house) or metaphorical (on time, with gusto, beside herself), and relations like marking the agent in Hamlet was written by Shakespeare.

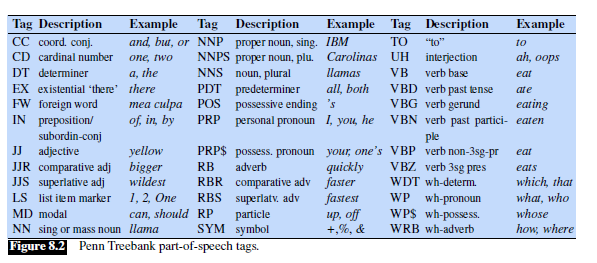
A particle resembles a preposition or an adverb and is used in combination with a verb. Particles often have extended meanings that aren’t quite the same as the prepositions they resemble, as in the particle over in she turned the paper over. A verb and a particle acting as a single unit is called a phrasal verb. The meaning of phrasal verbs is often non-compositional—not predictable from the individual meanings of the verb and the particle. Thus, turn down means ‘reject’, rule out ‘eliminate’, and go on ‘continue’. Determiners like this and that (this chapter, that page) can mark the start of an

article English noun phrase. Articles like a, an, and the, are a type of determiner that mark discourse properties of the noun and are quite frequent; the is the most common word in written English, with a and an right behind.

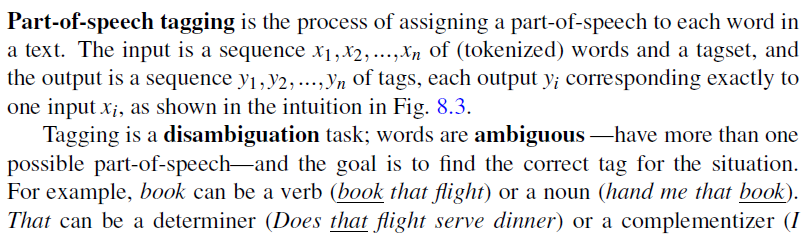
Conjunctions join two phrases, clauses, or sentences. Coordinating conjunctions like and, or, and but join two elements of equal status. Subordinating conjunctions are used when one of the elements has some embedded status. For example, the subordinating conjunction that in “I thought that you might like some milk” links the main clause I thought with the subordinate clause you might like some milk. This clause is called subordinate because this entire clause is the “content” of the main verb thought. Subordinating conjunctions like that which link a verb to its argument complementizer in this way are also called complementizers.

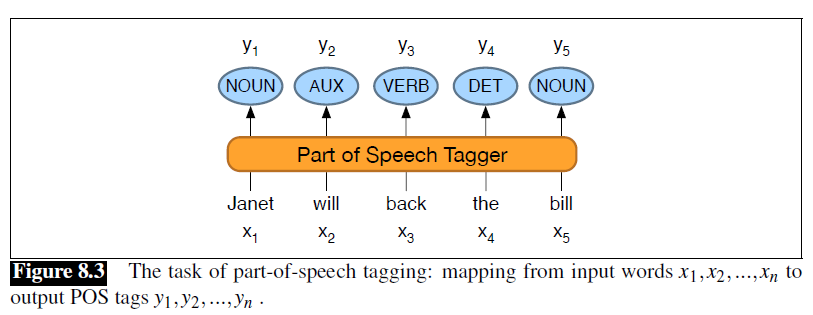
Pronouns act as a shorthand for referring to an entity or event. Personal pronouns refer to persons or entities (you, she, I, it, me, etc.). Possessive pronouns are forms of personal pronouns that indicate either actual possession or more often just an abstract relation between the person and some object (my, your, his, her, its, one’s, wh our, their). Wh-pronouns (what, who, whom, whoever) are used in certain question

Auxiliary verbs mark semantic features of a main verb such as its tense, whether it is completed (aspect), whether it is negated (polarity), and whether an action is necessary, possible, suggested, or desired (mood). English auxiliaries include the copula copula verb be, the two verbs do and have, forms, as well as modal verbs used to modal mark the mood associated with the event depicted by the main verb: can indicates ability or possibility, may permission or possibility, must necessity.

An English-specific tagset, the 45-tag Penn Treebank tagset (Marcus et al., 1993), shown in Fig. 8.2, has been used to label many syntactically annotated corpora like the Penn Treebank corpora, so is worth knowing about:

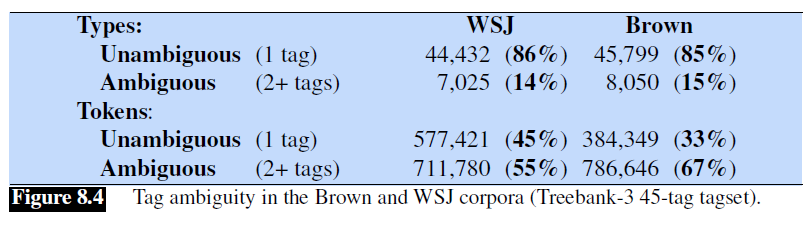
**Part-of-Speech Tagging**



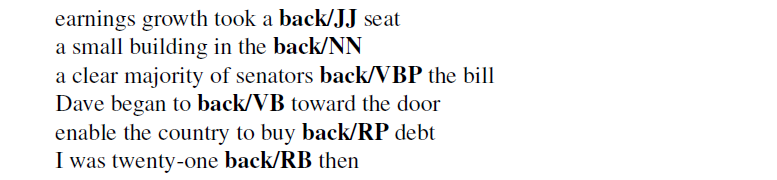


thought that your flight was earlier). The goal of POS-tagging is to resolve these ambiguities, choosing the proper tag for the context.

The accuracy of part-of-speech tagging algorithms (the percentage of test set tags that match human gold labels) is extremely high. One study found accuracies over 97% across 15 languages from the Universal Dependency (UD) treebank (Wu and Dredze, 2019). Accuracies on various English treebanks are also 97% (no matter the algorithm; HMMs, CRFs, BERT perform similarly). This 97% number is also about the human performance on this task, at least for English (Manning, 2011).



We’ll introduce algorithms for the task in the next few sections, but first let’s explore the task. Exactly how hard is it? Fig. 8.4 shows that most word types (85-86%) are unambiguous (Janet is always NNP, hesitantly is always RB). But the ambiguous words, though accounting for only 14-15% of the vocabulary, are very common, and 55-67% of word tokens in running text are ambiguous. Particularly ambiguous common words include that, back, down, put and set; here are some examples of the 6 different parts of speech for the word back:



Nonetheless, many words are easy to disambiguate, because their different tags aren’t equally likely. For example, a can be a determiner or the letter a, but the determiner sense is much more likely.

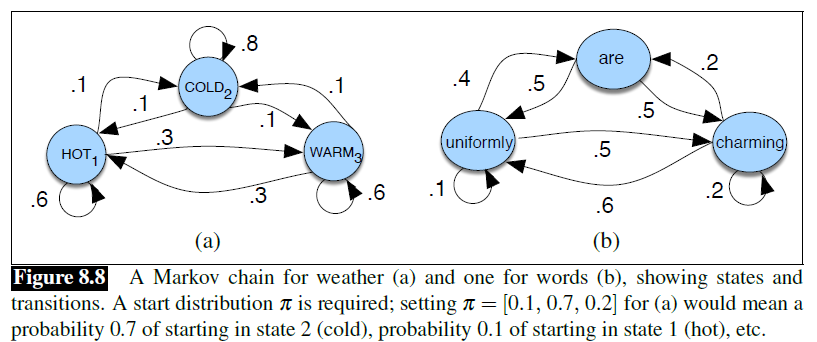
This idea suggests a useful baseline: given an ambiguous word, choose the tag which is most requent in the training corpus. This is a key concept:

**HMM Part-of-Speech Tagging**

An HMM is a probabilistic sequence model: given a sequence of units (words,letters, morphemes, sentences, whatever), it computes a probability distribution over possible sequences of labels and chooses the best label sequence.

**Markov Chains**

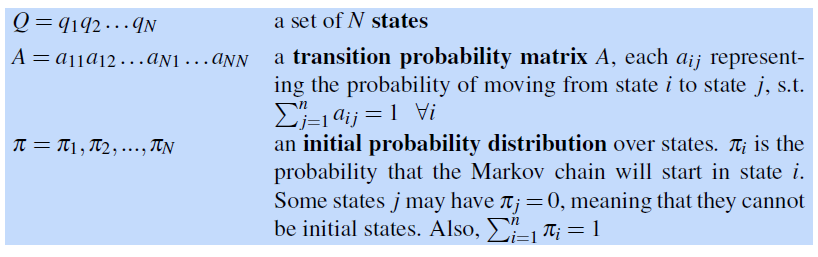
The HMM is based on augmenting the Markov chain. A Markov chain is a model that tells us something about the probabilities of sequences of random variables, states, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything, for example the weather. A Markov chain makes a very strong assumption that if we want to predict the future in the sequence, all that matters is the current state. All the states before the current state have no impact on the future except via the current state. It’s as if to predict tomorrow’s weather you could examine today’s weather but you weren’t allowed to look at yesterday’s weather.



More formally, consider a sequence of state variables . A Markov Markov model embodies theMarkov assumption on the probabilities of this sequence: that assumption when predicting the future, the past doesn’t matter, only the present.



Figure 8.8a shows a Markov chain for assigning a probability to a sequence of weather events, for which the vocabulary consists of HOT, COLD, and WARM. The states are represented as nodes in the graph, and the transitions, with their probabilities, as edges. The transitions are probabilities: the values of arcs leaving a given state must sum to 1. Figure 8.8b shows a Markov chain for assigning a probability to a sequence of words . This Markov chain should be familiar; in fact, it represents a bigram language model, with each edge expressing the probability ! Given the two models in Fig. 8.8, we can assign a probability to any sequence from our vocabulary.

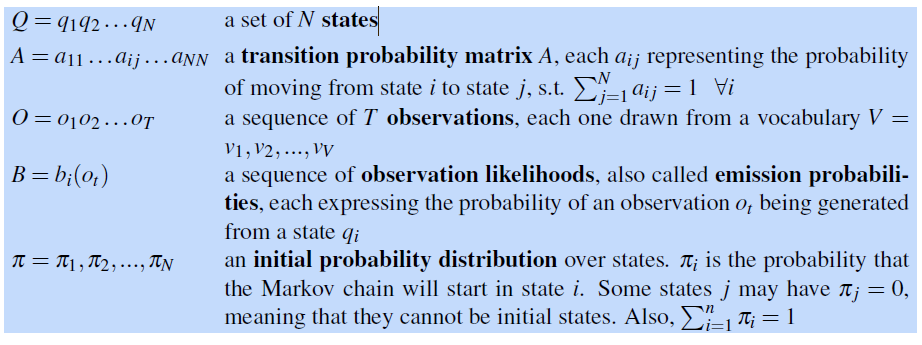


Before you go on, use the sample probabilities in Fig. 8.8a (with to compute the probability of each of the following sequences:

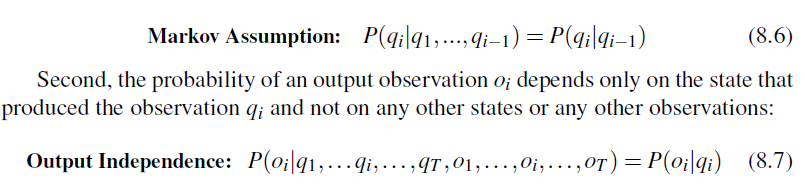
**The Hidden Markov Model**

A Markov chain is useful when we need to compute a probability for a sequence of observable events. In many cases, however, the events we are interested in are hidden: we don’t observe them directly. For example we don’t normally observe part-of-speech tags in a text. Rather, we see words, and must infer the tags from the word sequence. We call the tags hidden because they are not observed.

A hidden Markov model (HMM) allows us to talk about both observed events model (like words that we see in the input) and hidden events (like part-of-speech tags) that we think of as causal factors in our probabilistic model. An HMM is specified by the following components:



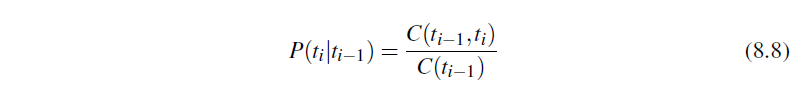
A first-order hidden Markov model instantiates two simplifying assumptions. First, as with a first-order Markov chain, the probability of a particular state depends only on the previous state:



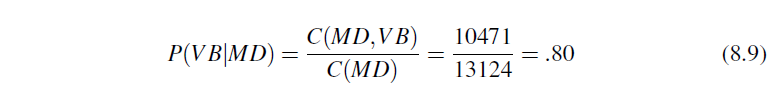
**The components of an HMM tagger**

Let’s start by looking at the pieces of an HMM tagger, and then we’ll see how to use it to tag. An HMM has two components, the A and B probabilities.

The A matrix contains the tag transition probabilities which represent the probability of a tag occurring given the previous tag. For example, modal verbs like will are very likely to be followed by a verb in the base form, a VB, like race, so we expect this probability to be high. We compute the maximum likelihood estimate of this transition probability by counting, out of the times we see the first tag in a labeled corpus, how often the first tag is followed by the second:



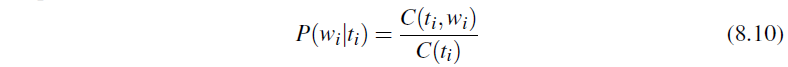
In the WSJ corpus, for example, MD occurs 13124 times of which it is followed by VB 10471, for an MLE estimate of



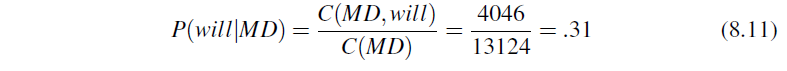
Let’s walk through an example, seeing how these probabilities are estimated and used in a sample tagging task, before we return to the algorithm for decoding.

In HMM tagging, the probabilities are estimated by counting on a tagged training corpus. For this example we’ll use the tagged WSJ corpus.

The B emission probabilities, , represent the probability, given a tag (say MD), that it will be associated with a given word (say will). The MLE of the emission probability is



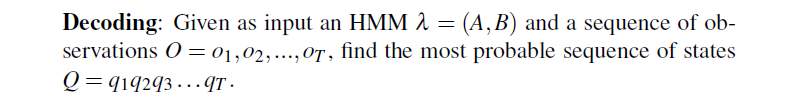
Of the 13124 occurrences of MD in the WSJ corpus, it is associated with will 4046 times:

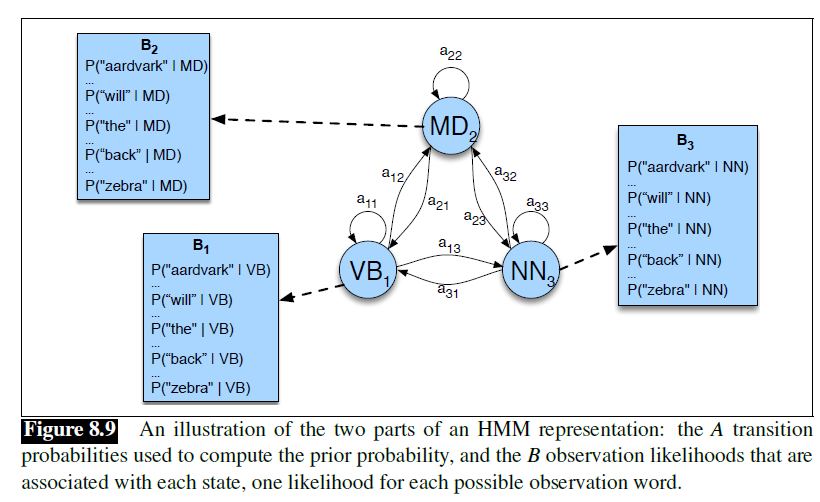


We saw this kind of Bayesian modeling in Chapter 4; recall that this likelihood term is not asking “which is the most likely tag for the word *will?*” That would be the posterior P(MD|will). Instead, P(will|MD) answers the slightly counterintuitive question “If we were going to generate a MD, how likely is it that this modal would be will?

**HMM tagging as decoding**

For any model, such as an HMM, that contains hidden variables, the task of determining the hidden variables sequence corresponding to the sequence of observations is called decoding. More formally,

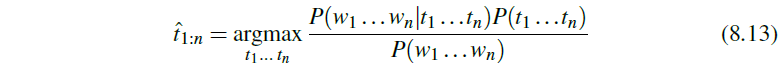




For part-of-speech tagging, the goal of HMM decoding is to choose the tag sequence that is most probable given the observation sequence of n words :



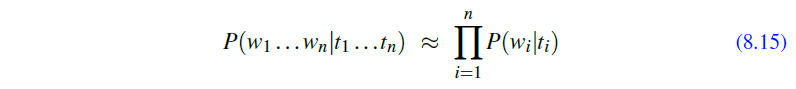
The way we’ll do this in the HMM is to use Bayes’ rule to instead compute:



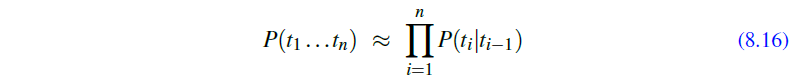
Furthermore, we simplify Eq. 8.13 by dropping the denominator



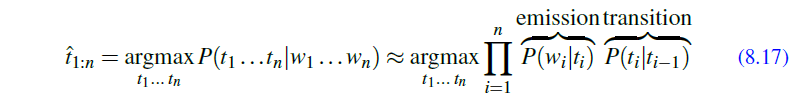
HMM taggers make two further simplifying assumptions. The first is that the probability of a word appearing depends only on its own tag and is independent of neighbouring words and tags:



The second assumption, the bigram assumption, is that the probability of a tag is dependent only on the previous tag, rather than the entire tag sequence;



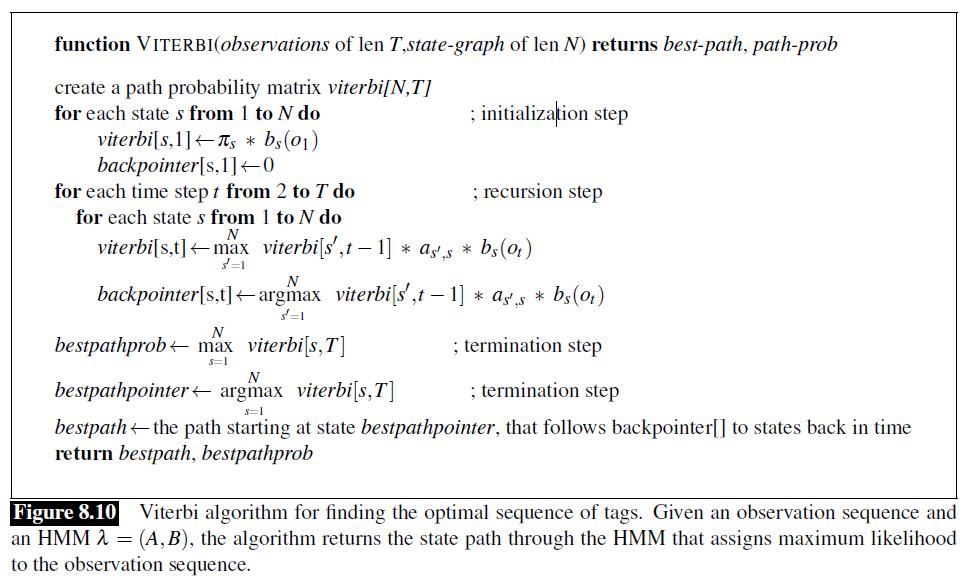
Plugging the simplifying assumptions from Eq. 8.15 and Eq. 8.16 into Eq. 8.14 results in the following equation for the most probable tag sequence from a bigram tagger:



The two parts of Eq. 8.17 correspond neatly to the B emission probability and A transition probability that we just defined above!

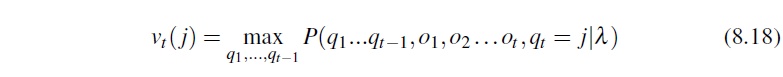
**The Viterbi Algorithm**

The decoding algorithm for HMMs is the Viterbi algorithm shown in Fig. 8.10. As an instance of dynamic programming, Viterbi resembles the dynamic programming minimum edit distance

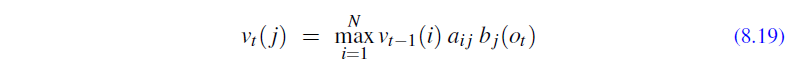


The Viterbi algorithm first sets up a probability matrix or lattice, with one column for each observation and one row for each state in the state graph. Each column thus has a cell for each state qi in the single combined automaton. Figure 8.11 shows an intuition of this lattice for the sentence Janet will back the bill.

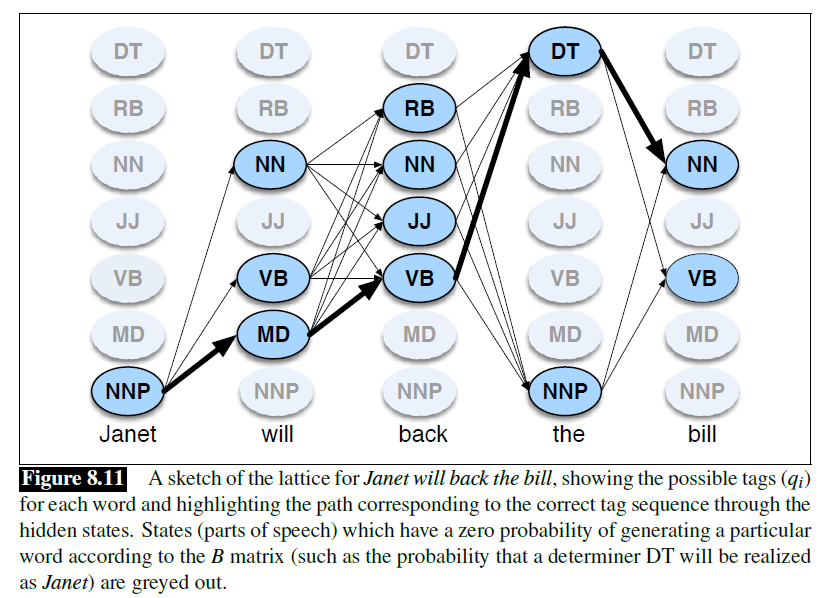
Each cell of the lattice, , represents the probability that the HMM is in state j after seeing the first t observations and passing through the most probable state sequence , given the HMM . The value of each cell , is computed by recursively taking the most probable path that could lead us to this cell. Formally, each cell expresses the probability

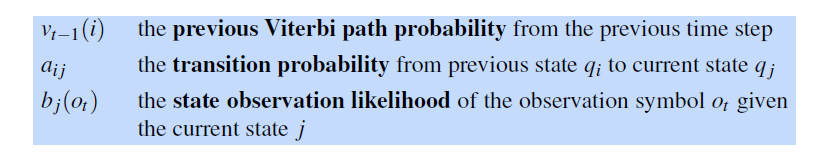


We represent the most probable path by taking the maximum over all possible previous state sequences . Like other dynamic programming algorithms, Viterbi fills each cell recursively. Given that we had already computed the probability of being in every state at time t􀀀1, we compute the Viterbi probability by taking the most probable of the extensions of the paths that lead to the current cell. For a given state qj at time t, the value , is computed as



The three factors that are multiplied in Eq. 8.19 for extending the previous paths to compute the Viterbi probability at time t are





**Baum-Welch algorithm**

Also known as the forward-backward algorithm, the Baum-Welch algorithm is a dynamic programming approach and a special case of the expectation-maximization algorithm (EM algorithm). Its purpose is to tune the parameters of the HMM, namely the state transition matrix A, the emission matrix B, and the initial state distribution π₀, such that the model is maximally like the observed data

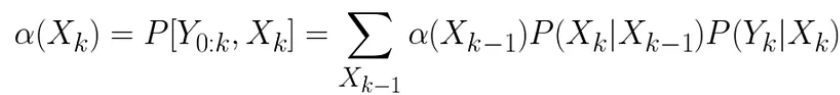
There are a few phases for this algorithm, including the initial phase, the forward phase, the backward phase, and the update phase. The forward and the backward phase form the E-step of the EM algorithm, while the update phase itself is the M-step.

**Initial phase**

In the initial phase, the content of the parameter matrices A, B, π₀ are initialized, and it could be done randomly if there is no prior knowledge about them.

**Forward phase**

In the forward phase, the following recursive alpha function is calculated. For the deviation of the function, I would strongly recommend this YouTube video as the speaker presented it clearly and explained it very well.



There are a few points to make here:

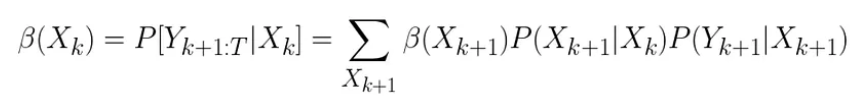
1. The alpha function is defined as the joint probability of the observed data up to time k and the state at time k
2. It is a recursive function because the alpha function appears in the first term of the right hand side (R.H.S.) of the equation, meaning that the previous alpha is reused in the calculation of the next. This is also why it is called the forward phase.
3. The second term of the R.H.S. is the state transition probability from A, while the last term is the emission probability from B.
4. The R.H.S. is summed over all possible states at time k -1.

It should be pointed out that, each alpha contains the information from the observed data up to time k, and to get the next alpha, we only need to reuse the current alpha, and add information about the transition to the next state and the next observed variable. This recursive behavior saves computations of getting the next alpha by freeing us from looking through the past observed data every time.

By the way, we need the following starting alpha to begin the recursion.

**Backward phase**

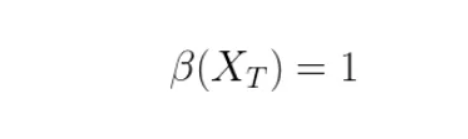
In this phase we have to follow the following formula.



Similar points could be made here:

1. The beta function is defined as the conditional probability of the observed data from time k+1 given the state at time k
2. It is a recursive function because the beta function appears in first term of the right hand side of the equation, meaning that the next beta is reused in the calculation of the current one. This is also why it is called a backward phase.
3. The second term of the R.H.S. is the state transition probability from A, while the last term is the emission probability from B.
4. The R.H.S. is summed over all possible states at time k +1.

Again, we need the ending beta to start the recursion.



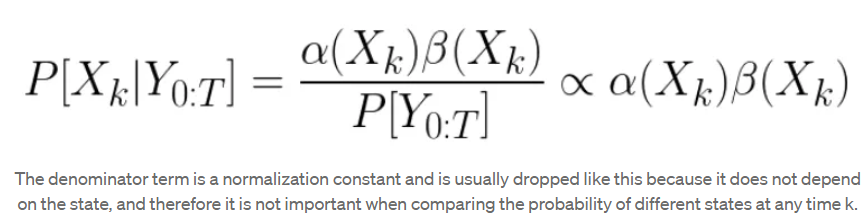
Why the alpha and the beta functions?

Firstly, as mentioned, they are both recursive functions, which means that we could reuse previous answer as the input for the next answer. This is what dynamic programming is about — you could save time by reusing old result!

Secondly, the formula in the forward phase is very useful. Suppose you have a set of well-trained transition and emission parameters, and given that your problem is to, in real-time, find out the mysterious hidden truth from observed data. Then you actually could do it like this! When you get one data point (data point p), then you could put it into the formula which will give you the probability distribution of the associated hidden state, and from which you could pick the most probable one as your answer. And the story does not stop here, as you get the next data point (data point q), and you put it again into the formula, it will give you another probability distribution for you to pick the best choice, but this is not only based on data point q and the transition and emission parameters, but also the data point p. Such use of the formula is called filtering.

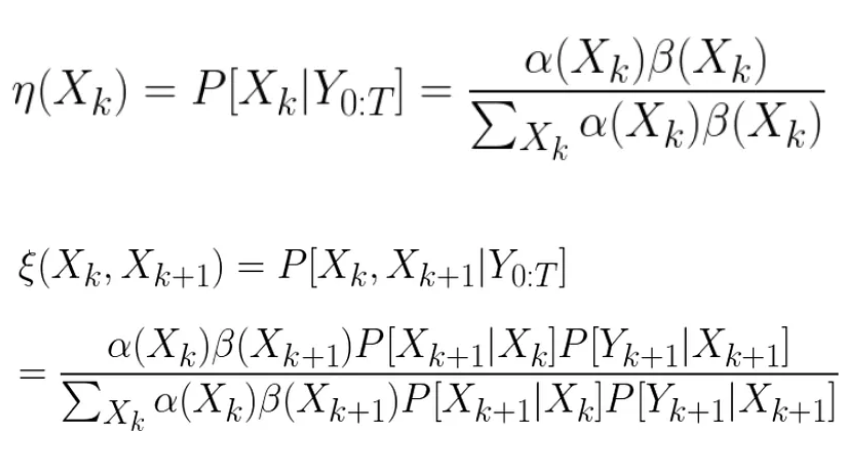
Thirdly, and continuing the above discussion, that suppose you collected many data points already, and because you know that the earlier the data point, the less observed data the choice of your answer based on. Therefore you would like to improve that by somehow ‘injecting’ information from the later data into the earlier ones. This is where the backward formula comes into play. Such use of the formula is called smoothing.

Fourthly, this is about the combination of the last two paragraphs. With the help of the alpha and the beta formula, one could determine the probability distribution of the state variable at any time k given the whole sequence of observed data. This could also be understood mathematically.



Lastly, the result from the alpha and the beta functions are useful in the update phase.

Update phase



For the deviation of the above formulas, if you have watched the YouTube videos that I suggested for the forward and backward formula, and you can understand them, then probably you will have no problem to derive these two yourself.

The first formula here is just repeating what we have seen above, and to recap, it is to tell us the probability distribution of a state at time k given all observed data we have. The second formula, however, tells us a bit different thing which is the joint probability of two consecutive states given the data. They make use of the alpha function, the beta function, the transition and the emission that are already available. These two formulas are further used to finally do the update.



It was mentioned that the Baum-Welch algorithm is a case of EM algorithm. Here I will explain why very briefly. The alpha and the beta function form the E-step because they predict for the expected hidden states given the observed data and the parameter matrices A, B, π₀. The update phase is the M-step because the last three update formulas are so derived that the L.H.S. parameters will best fit the expected hidden states given the observed data.

The Baum-Welch algorithm is a case of EM algorithm that, in the E-step, the forward and the backward formulas tell us the expected hidden states given the observed data and the set of parameter matrices before-tuned. The M-step update formulas then tune the parameter matrices to best fit the observed data and the expected hidden states. And these two steps are then iterated over and over again until the parameters converged, or until the model has reached some certain accuracy requirement.

Like any machine learning algorithm, this algorithm could be overfitting the data, as by definition the M-step encourages the model to approach the observed data as good as possible. Also, although we have not talked too much about the initial phase, it indeed affects the final performance of the model (as a problem of trapping the model in local optimum), so one might want to try different ways of initializing the parameters and see what works better.

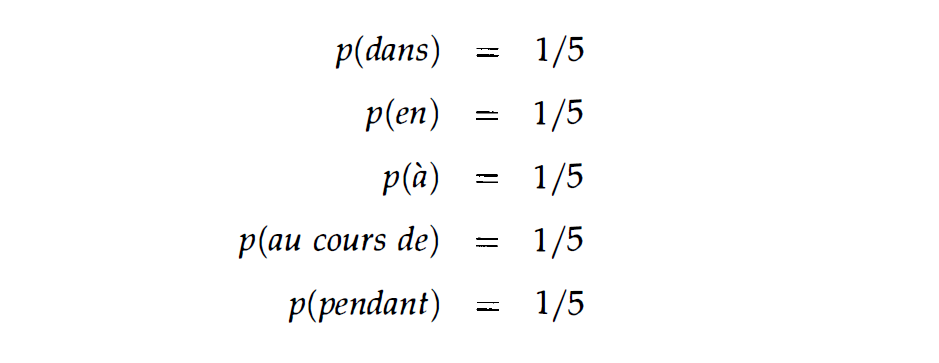
**Maximum Entropy Models:**

Maximum entropy probability models offer a clean way to combine diverse pieces of contextual evidence in order to estimate the probability of a certain linguistic class occurring with a certain linguistic context. Suppose we wish to model an expert translator's decisions concerning the proper French rendering of the English word in. Our model p of the expert's decisions assigns to each French word or phrase fan estimate, p(f), of the probability that the expert would choose fas a translation of in. To guide us in developing p, we collect a large sample of instances of the expert's decisions. Our goal is to extract a set of facts about the decision-making process from the sample (the first task of modeling) that will aid us in constructing a model of this process (the second task).

One obvious clue we might glean from the sample is the list of allowed trans­lations. For example, we might discover that the expert translator always chooses among the following five French phrases: {dans, en, a, au cours de, pendant}. With this information in hand, we can impose our first constraint on our model p:

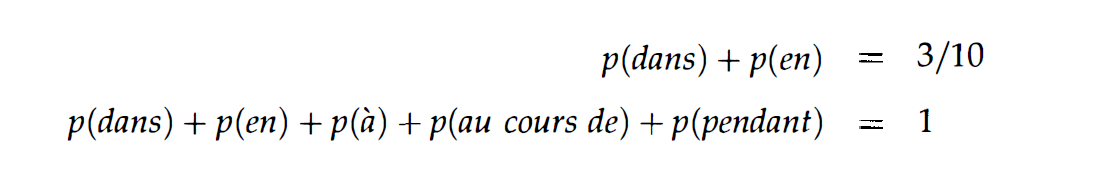
p(dans) + p(en) + p(a) + p(au cours de)+ p(pendant) = 1

This equation represents our first statistic of the process; we can now proceed to search for a suitable model that obeys this equation. Of course, there are an infinite number of models p for which this identity holds. One model satisfying the above equation is p(dans) = 1; in other words, the model always predicts dans. Another model obeying this constraint predicts pendant with a probability of 1 /2, and a with a probability of 1/2. But both of these models offend our sensibilities: knowing only that the expert always chose from among these five French phrases, how can we justify either of these probability distributions? Each seems to be making rather bold assump­tions, with no empirical justification. Put another way, these two models assume more than we actually know about the expert's decision-making process. All we know is that the expert chose exclusively from among these five French phrases; given this the most intuitively appealing model is the following:

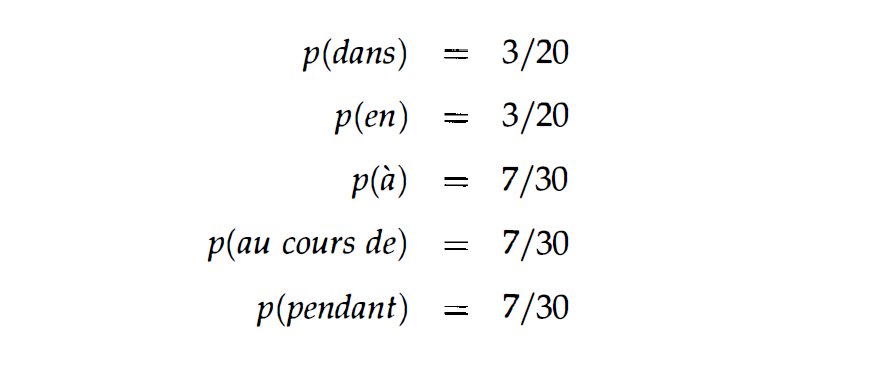


This model, which allocates the total probability evenly among the five possible phrases, is the most uniform model subject to our knowledge. It is not, however, the most uniform overall; that model would grant an equal probability to every possible French phrase.

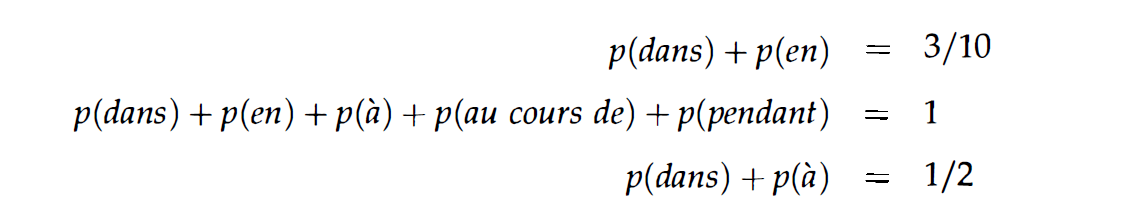
We might hope to glean more clues about the expert's decisions from our sample. Suppose we notice that the expert chose either dans or en 30% of the time. We could apply this knowledge to update our model of the translation process by requiring that p satisfy two constraints:



Once again there are many probability distributions consistent with these two constraints. In the absence of any other knowledge, a reasonable choice for p is again the most uniform-that is, the distribution which allocates its probability as evenly as possible, subject to the constraints:



Say we inspect the data once more, and this time notice another interesting fact: in half the cases, the expert chose either dans or a. We can incorporate this information into our model as a third constraint:



We can once again look for the most uniform p satisfying these constraints, but now the choice is not as obvious. As we have added complexity, we have encountered two difficulties at once. First, what exactly is meant by "uniform," and how can we measure the uniformity of a model? Second, having determined a suitable answer to these questions, how do we go about finding the most uniform model subject to a set of constraints like those we have described?

**Maximum Entropy Modelling**

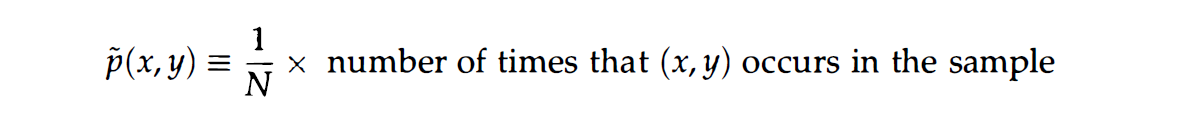
We consider a random process that produces an output value y, a member of a finite set Y. For the translation example just considered, the process generates a translation of the word in, and the output y can be any word in the set {dans, en, a, au cours de, pendant}. In generating y, the process may be influenced by some contextual information x, a member of a finite set X. In the present example, this information could include the words in the English sentence surrounding in.

Our task is to construct a stochastic model that accurately represents the behavior of the random process. Such a model is a method of estimating the conditional prob­ability that, given a context x, the process will output y. We will denote by p(y\x) the probability that the model assigns to y in context x. With a slight abuse of notation, we will also use p(y\x) to denote the entire conditional probability distribution provided by the model, with the interpretation that y and x are placeholders rather than specific instantiations. The proper interpretation should be clear from the context. We will de­note by P the set of all conditional probability distributions. Thus a model p(y\x) is, by definition, just an element of P.

**Training Data**

To study the process, we observe the behavior of the random process for some time, collecting a large number of samples (x1,y1), (x2,Y2), ... , (xN,YN)· In the example we have been considering, each sample would consist of a phrase x containing the words surrounding in, together with the translation y of in that the process produced. For now, we can imagine that these training samples have been generated by a human expert who was presented with a number of random phrases containing in and asked to choose a good translation for each.

We can summarize the training sample in terms of its empirical probability distri­bution p, defined by

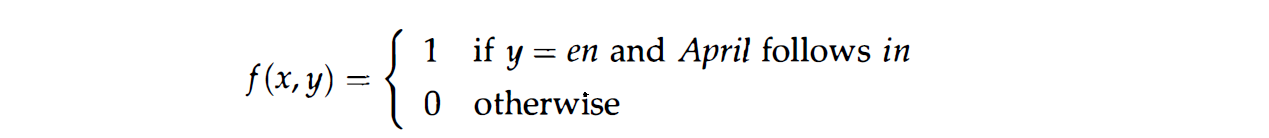


Typically, a particular pair (x,y) will either not occur at all in the sample, or will occur at most a few times.

**Statistics, Features and Constraints:**

Our goal is to construct a statistical model of the process that generated the training sample p(x, y). The building blocks of this model will be a set of statistics of the training sample. In the current example we have employed several such statistics: the frequency with which in translated to either dans or en was 3/10; the frequency with which it translated to either dans or au cours de was 1/2; and so on. These particular statistics were independent of the context, but we could also consider statistics that depend on the conditioning information x. For instance, we might notice that, in the training sample, if April is the word following in, then the translation of in is en with frequency 9/10.

To express the fact that in translates as en when April is the following word, we can introduce the indicator function:

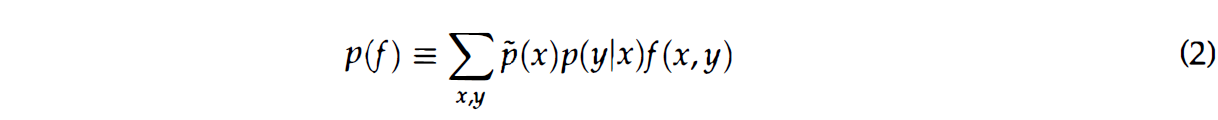


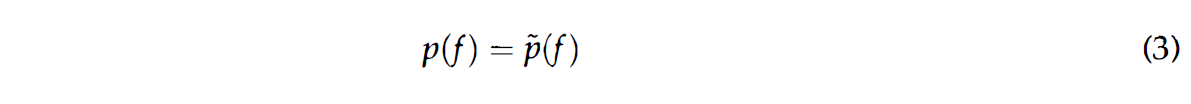
The expected value off with respect to the empirical distribution p(x, y) is exactly the statistic we are interested in. We denote this expected value by



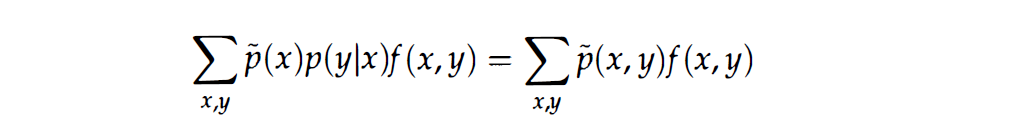
We can express any statistic of the sample as the expected value of an appropriate binary-valued indicator function f. We call such function a feature function or feature for short. (As with probability distributions, we will sometimes abuse notation and use f (x, y) to denote both the value off at a particular pair (x, y) as well as the entire function f.)

When we discover a statistic that we feel is useful, we can acknowledge its im­portance by requiring that our model accord with it. We do this by constraining the expected value that the model assigns to the corresponding feature function f. The expected value off with respect to the model p(ylx) is





Combining (1), (2) and (3) yields the more explicit equation



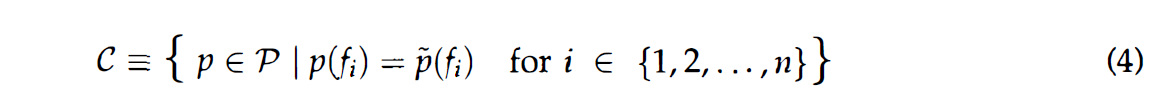
We call the requirement (3) a constraint equation or simply a constraint. By re­stricting attention to those models p(ylx) for which (3) holds, we are eliminating from consideration those models that do not agree with the training sample on how often the output of the process should exhibit the feature f.

To sum up so far, we now have a means of representing statistical phenomena inherent in a sample of data (namely, p(f)), and also a means of requiring that our model of the process exhibit these phenomena (namely, p(f) = p(f)).

One final note about features and constraints bears repeating: although the words "feature" and "constraint" are often used interchangeably in discussions of maximum entropy, we will be vigilant in distinguishing the two and urge the reader to do likewise. A feature is a binary-valued function of (x, y); a constraint is an equation between the expected value of the feature function in the model and its expected value in the training data.

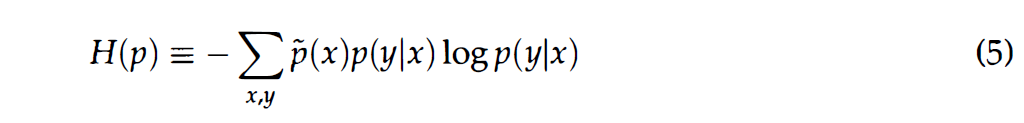
**The Maximum Entropy Principle**

Suppose that we are given n feature functions fi, which determine statistics we feel are important in modeling the process. We would like our model to accord with these statistics. That is, we would like p to lie in the subset C of P defined by



Here P is the space of all (unconditional) probability distributions on three points, sometimes called a simplex. If we impose no constraints (depicted in (a)), then all probability models are allowable. Imposing one linear constraint C1 restricts us to those p E P that lie on the region defined by C1, as shown in (b). A second linear constraint could determine p exactly, if the two constraints are satisfiable; this is the case in (c), where the intersection of C1 and C2 is non-empty. Alternatively, a second linear constraint could be inconsistent with the first-for instance, the first might require that the probability of the first point is 1 /3 and the second that the probability of the third point is 3/4-this is shown in (d). In the present setting, however, the linear constraints are extracted from the training sample and cannot, by construction, be inconsistent. Furthermore, the linear constraints in our applications will not even come close to determining p E P uniquely as they do in (c); instead, the set of allowable models will be infinite.

Among the models p EC, the maximum entropy philosophy dictates that we select the most uniform distribution. But now we face a question left open in Section 2: what does "uniform" mean? A mathematical measure of the uniformity of a conditional distribution p(ylx) is provided by the conditional entropy1



**Conditional Random Fields**

While the HMM is a useful and powerful model, it turns out that HMMs need a number of augmentations to achieve high accuracy. For example, in POS tagging unknown as in other tasks, we often run into unknown words: proper names and acronyms words are created very often, and even new common nouns and verbs enter the language at a surprising rate. It would be great to have ways to add arbitrary features to help with this, perhaps based on capitalization or morphology (words starting with capital letters are likely to be proper nouns, words ending with -ed tend to be past tense (VBD or VBN), etc.) Or knowing the previous or following words might be a useful feature (if the previous word is the, the current tag is unlikely to be a verb).

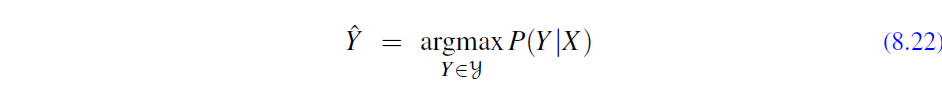
There is a discriminative sequence model based on log-linear models: CRF the conditional random field (CRF). We’ll describe here the linear chain CRF, the version of the CRF most commonly used for language processing, and the one whose conditioning closely matches the HMM.

Assuming we have a sequence of input words and want to compute a sequence of output tags .In an HMM to compute the best tag sequence that maximizes P(Y|X ) we rely on Bayes’ rule and the likelihood P(X|Y ):



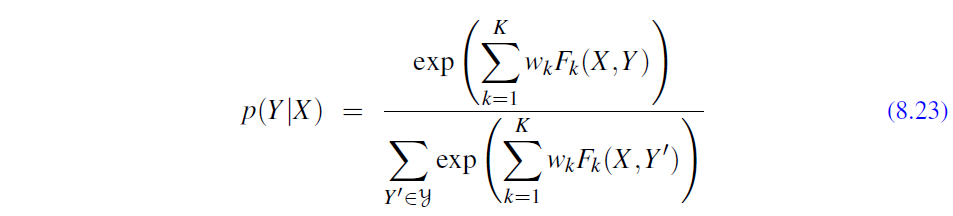
In a CRF, by contrast, we compute the posterior p(Y|X) directly, training the CRF

to discriminate among the possible tag sequences:

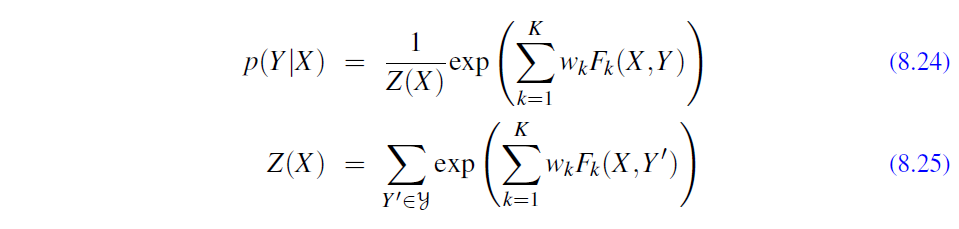


However, the CRF does not compute a probability for each tag at each time step. Instead, at each time step the CRF computes log-linear functions over a set of relevant features, and these local features are aggregated and normalized to produce a global probability for the whole sequence.

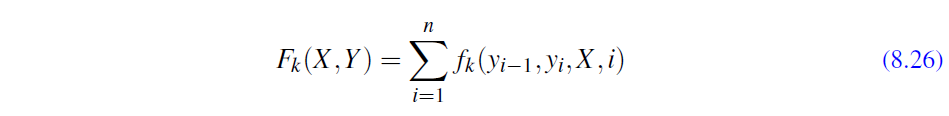
Let’s introduce the CRF more formally, again using X and Y as the input and output sequences. A CRF is a log-linear model that assigns a probability to an entire output (tag) sequence Y, out of all possible sequences Y, given the entire input (word) sequence X. We can think of a CRF as like a giant version of what multinomial logistic regression does for a single token. Recall that the feature function f in regular multinomial logistic regression can be viewed as a function of a tuple: a token x and a label y (page 89). In a CRF, the function F maps an entire input sequence X and an entire output sequence Y to a feature vector. Let’s assume we have K features, with a weight for each feature :



It’s common to also describe the same equation by pulling out the denominator into a function Z(X):



We’ll call these K functions global features, since each one is a property of the entire input sequence X and output sequence Y. We compute them by decomposing into a sum of local features for each position i in Y:

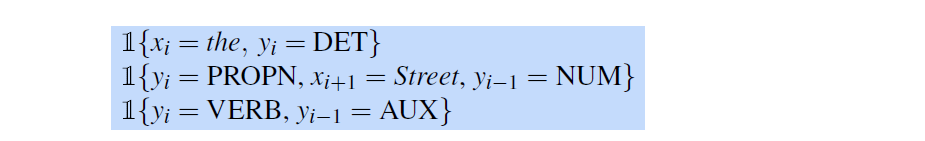


Each of these local features in a linear-chain CRF is allowed to make use of the current output token , the previous output token , the entire input string X (or any subpart of it), and the current position i. This constraint to only depend on the current and previous output tokens and are what characterizes a linear linear chain chain CRF. As we will see, this limitation makes it possible to use versions of the CRF efficient Viterbi and Forward-Backwards algorithms from the HMM. A general CRF, by contrast, allows a feature to make use of any output token, and are thus necessary for tasks in which the decision depend on distant output tokens, like . General CRFs require more complex inference, and are less commonly used for language processing.

**Features in a CRF POS Tagger**

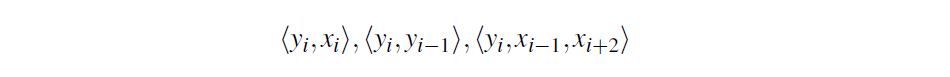
Let’s look at some of these features in detail, since the reason to use a discriminative sequence model is that it’s easier to incorporate a lot of features

Again, in a linear-chain CRF, each local feature at position i can depend on any information from: (,X, i). So some legal features representing common situations might be the following:

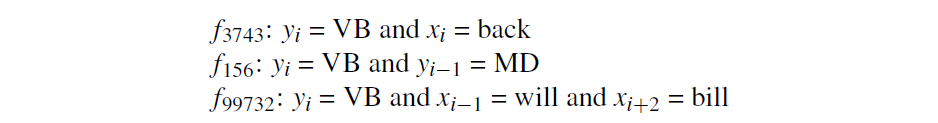


For simplicity, we’ll assume all CRF features take on the value 1 or 0. Above, we explicitly use the notation 1fxg to mean “1 if x is true, and 0 otherwise”. From now on, we’ll leave off the 1 when we define features, but you can assume each feature has it there implicitly.

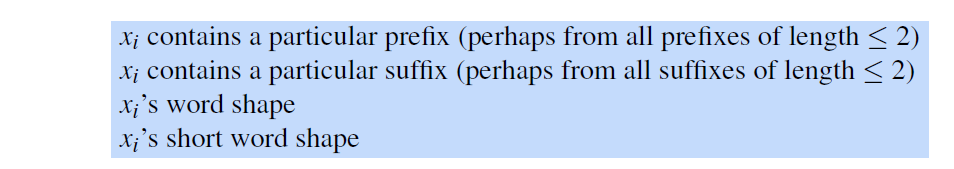
Although the idea of what features to use is done by the system designer by hand, the specific features are automatically populated by using feature templates as we briefly mentioned in Chapter 5. Here are some templates that only use information from (,X, i)



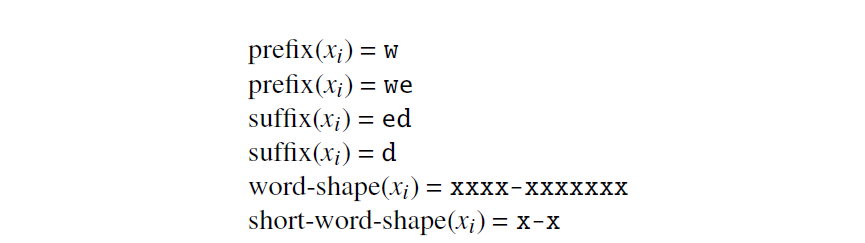
These templates automatically populate the set of features from every instance in the training and test set. Thus for our example Janet/NNP will/MD back/VB the/DT bill/NN, when xi is the word back, the following features would be generated and have the value 1 (we’ve assigned them arbitrary feature numbers):



It’s also important to have features that help with unknown words. One of the word important is word shape features, which represent the abstract letter pattern of the word by mapping lower-case letters to ‘x’, upper-case to ‘X’, numbers to ’d’, and retaining punctuation. Thus for example I.M.F. would map to X.X.X. and DC10-30 would map to XXdd-dd. A second class of shorter word shape features is also used. In these features consecutive character types are removed, so words in all caps map to X, words with initial-caps map to Xx, DC10-30 would be mapped to Xd-d but I.M.F would still map to X.X.X. Prefix and suffix features are also useful. In summary, here are some sample feature templates that help with unknown words:



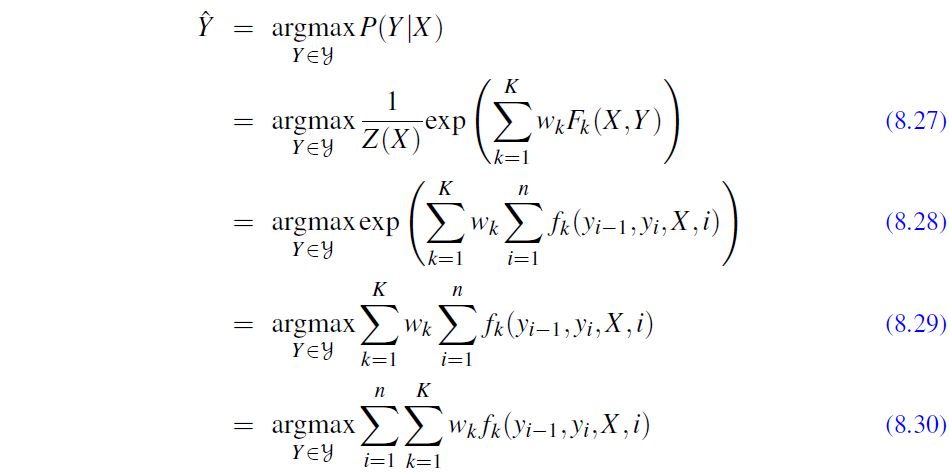
For example the word well-dressed might generate the following non-zero valued feature values:



The known-word templates are computed for every word seen in the training set; the unknown word features can also be computed for all words in training, or only on training words whose frequency is below some threshold. The result of the known-word templates and word-signature features is a very large set of features. Generally a feature cutoff is used in which features are thrown out if they have count < 5 in the training set.

**Inference and Training for CRFs**

How do we find the best tag sequence ˆY for a given input X? We start with Eq. 8.22:

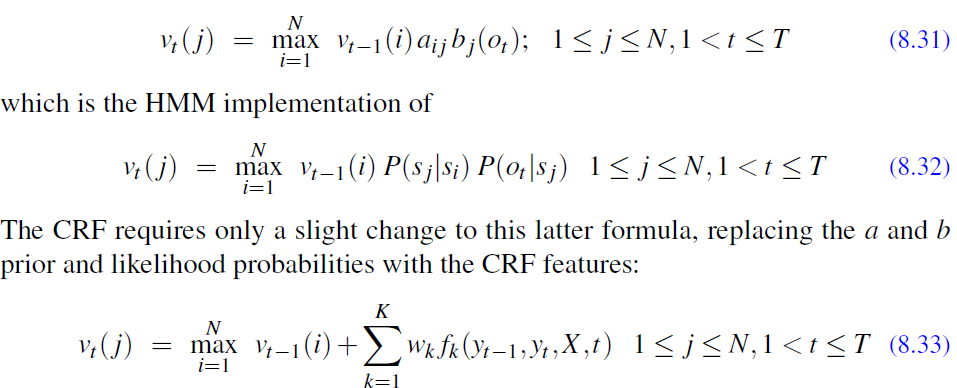


We can ignore the exp function and the denominator Z(X), as we do above, because exp doesn’t change the argmax, and the denominator Z(X) is constant for a given observation sequence X.

How should we decode to find this optimal tag sequence ˆ y? Just as with HMMs, we’ll turn to the Viterbi algorithm, which works because, like the HMM, the linearchain CRF depends at each timestep on only one previous output token .

Concretely, this involves filling an NT array with the appropriate values, maintaining backpointers as we proceed. As with HMM Viterbi, when the table is filled, we simply follow pointers back from the maximum value in the final column to retrieve the desire.

The requisite changes from HMM Viterbi have to do only with how we fill each cell. Recall from Eq. 8.19 that the recursive step of the Viterbi equation computes the Viterbi value of time t for state j asd set of labels.



Learning in CRFs relies on the same supervised learning algorithms we presented for logistic regression. Given a sequence of observations, feature functions, and corresponding outputs, we use stochastic gradient descent to train the weights to maximize the log-likelihood of the training corpus. The local nature of linear-chain CRFs means that the forward-backward algorithm introduced for HMMs in Appendix A can be extended to a CRF version that will efficiently compute the necessary derivatives. As with logistic regression, L1 or L2 regularization is important.

**Important Question:**

1. Implement the “most likely tag” baseline. Find a POS-tagged training set, and use it to compute for each word the tag that maximizes p(t|w). You will need to implement a simple tokenizer to deal with sentence boundaries. Start by assuming that all unknown words are NN and compute your error rate on known and unknown words. Now write at least five rules to do a better job of tagging unknown words, and show the difference in error rates.
2. Build a bigram HMM tagger. You will need a part-of-speech-tagged corpus.First split the corpus into a training set and test set. From the labeled training set, train the transition and observation probabilities of the HMM tagger directly on the hand-tagged data. Then implement the Viterbi algorithm so youcan decode a test sentence. Now run your algorithm on the test set. Report its error rate and compare its performance to the most frequent tag baseline. Names of works of art (books, movies, video games, etc.) are quite different
3. from the kinds of named entities we’ve discussed in this chapter. Collect a list of names of works of art from a particular category from a Web-based source (e.g., gutenberg.org, amazon.com, imdb.com, etc.). Analyze your list and give examples of ways that the names in it are likely to be problematic for the techniques described in this chapter.
4. Analyze the components of HMM tagger.
5. What is POS tagging? Explaing How POS tagging is an aimportant application of NLP.

**Project :**

*Build a system to correctly identify different parts of speech to detect gender of a word from a corpus.*

**Interview Questions;**

1. What Is Part Of Speech (pos) Tagging?
2. What is Hidden Markov Model?
3. How HMM helps in POS tagging?
4. What is Conditional Random Fields?
5. How CRF can be used in entity name tagging?

**Research Papers:**

1. Song, S., Zhang, N., & Huang, H. (2019). Named entity recognition based on conditional random fields. Cluster Computing, 22, 5195-5206.
2. Liu, Z., Tang, B., Wang, X., & Chen, Q. (2017). De-identification of clinical notes via recurrent neural network and conditional random field. Journal of biomedical informatics, 75, S34-S42.
3. Suleiman, D., Awajan, A., & Al Etaiwi, W. (2017). The use of hidden Markov model in natural arabic language processing: A survey. Procedia computer science, 113, 240-247.
4. Paul, A., Purkayastha, B. S., & Sarkar, S. (2015, September). Hidden Markov model based part of speech tagging for Nepali language. In 2015 international symposium on advanced computing and communication (isacc) (pp. 149-156). IEEE.
5. Sun, S., Liu, H., Lin, H., & Abraham, A. (2012, October). Twitter part-of-speech tagging using pre-classification Hidden Markov model. In 2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC) (pp. 1118-1123). IEEE.

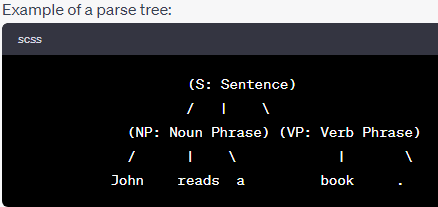
**UNIT - III**

**Syntax Parsing:**

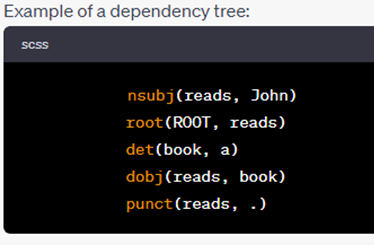
Syntax parsing, also known as syntactic parsing or parsing, is a fundamental task in Natural Language Processing (NLP) that involves analyzing the grammatical structure of a sentence to understand its syntactic relationships and hierarchical organization. The goal of syntax parsing is to parse a sentence into a structured representation, typically a parse tree or dependency tree, that illustrates the grammatical structure and the relationships between words in the sentence.

**There are two main types of syntax parsing approaches:**

Constituency Parsing: Constituency parsing involves dividing a sentence into a set of constituents (phrases) and representing the hierarchical relationships between these constituents. The result is typically a parse tree, where the root represents the whole sentence, and the internal nodes represent constituents, such as noun phrases, verb phrases, prepositional phrases, etc. The leaf nodes correspond to individual words in the sentence.



Dependency Parsing: Dependency parsing represents the grammatical relationships between words in a sentence using directed arcs. Each word in the sentence is a node in the tree, and the arcs represent the syntactic dependencies between words, showing which words are dependent on others. The root of the tree is usually an artificial node representing the root of the sentence.



Dependency parsing is often preferred in many modern NLP applications because it provides a more compact representation of the sentence's syntactic structure and is efficient for language understanding tasks like information extraction, named entity recognition, and relation extraction.

Syntax parsing can be accomplished using various algorithms and techniques, including rule-based approaches, statistical parsing models, and neural network-based methods. Data-driven approaches that learn from annotated parsing data have gained significant popularity due to their ability to generalize well to unseen sentences.

In recent years, deep learning techniques, particularly transformer-based models like BERT and GPT, have achieved state-of-the-art performance in syntax parsing, making it a crucial component in various NLP tasks and applications.

**Syntax CKY:**

Syntax CKY (Cocke-Kasami-Younger) is a parsing algorithm used in Natural Language Processing (NLP) for syntactic parsing, specifically for constituency parsing. The CKY algorithm is a dynamic programming approach that efficiently builds a parse tree for a given sentence based on a context-free grammar.

The context-free grammar used in CKY is typically in Chomsky Normal Form (CNF), where each production rule is of the form:

1. A -> B C (binary rule)
2. A -> "word" (terminal rule)

Here, A, B, and C are non-terminal symbols representing constituents (phrases), and "word" represents individual words in the sentence.

The CKY algorithm works by filling up a table, called the CKY table, for all possible subphrases of the sentence. The rows of the table represent the starting positions of the subphrases, and the columns represent the ending positions. Each cell of the table stores the constituents that span the corresponding subphrase.

**The steps of the CKY algorithm are as follows:**

1. Initialization: Fill the diagonal cells of the CKY table with the terminal rules corresponding to the individual words in the sentence.
2. Filling the Table: Traverse the CKY table in a diagonal manner, filling the cells with binary rules that combine constituents to build larger constituents. The constituents in the cells are determined based on the grammar rules and the constituents in the adjacent cells.
3. Backtracking: Once the CKY table is filled, the parse tree can be constructed by backtracking through the table and finding the constituents that span the entire sentence.

The CKY algorithm efficiently explores all possible parse trees for the given sentence and grammar, eliminating the need to enumerate all possible combinations explicitly. This makes it an efficient parsing method, especially for CNF grammars.

The CKY algorithm is widely used in parsing tasks where constituency-based parse trees are required. It has been employed in various NLP applications, such as information extraction, question answering, and natural language understanding. However, it should be noted that modern neural network-based approaches, such as shift-reduce and graph-based parsers, have gained popularity due to their ability to handle more complex structures and achieve better performance on large-scale datasets.

PCFG stands for Probabilistic Context-Free Grammar, and it is a formalism used in Natural Language Processing (NLP) for modeling the syntax of a language with probabilities. PCFGs extend context-free grammars by assigning probabilities to each production rule, indicating the likelihood of generating a particular phrase or constituent in a sentence.

A context-free grammar is a set of production rules that describe how sentences can be generated in a language. Each production rule consists of a non-terminal symbol on the left-hand side (LHS) and a sequence of symbols (non-terminals and/or terminals) on the right-hand side (RHS). Non-terminals are placeholders for constituents or phrases, while terminals represent actual words in the language.

**PCFGs:**

In a PCFG, each production rule is associated with a probability, denoting the likelihood of using that rule during the generation process. The probabilities must satisfy certain conditions, such as being non-negative and summing to one for each non-terminal symbol.

Formally, a PCFG is defined as a 4-tuple (N, Σ, R, P), where:

* N is a set of non-terminal symbols.
* Σ is a set of terminal symbols (words).
* R is a set of production rules of the form A -> β, where A is a non-terminal in N, and β is a sequence of symbols in (N ∪ Σ).
* P is a set of probabilities P(A -> β), where A -> β is a production rule, and P(A -> β) is the probability of using that rule.

The probabilities in a PCFG are typically estimated from a large corpus of annotated sentences using techniques such as maximum likelihood estimation or expectation-maximization.

PCFGs are commonly used in syntax parsing tasks, especially for constituency parsing. The probabilistic nature of PCFGs allows them to capture the most likely syntactic structures of sentences. During parsing, the goal is to find the most probable parse tree (constituency-based) or dependency tree (dependency-based) for a given sentence, given the PCFG.

PCFGs have been used in various NLP applications, such as machine translation, speech recognition, and information extraction. However, like other grammar-based approaches, PCFGs have some limitations in handling long-range dependencies and capturing semantic information. As a result, modern NLP techniques, particularly neural network-based models, have become more popular due to their ability to handle more complex language structures and learn from large amounts of data without relying on manually crafted grammars.

**PCFGs Inside:**

In Natural Language Processing (NLP), Probabilistic Context-Free Grammars (PCFGs) play a crucial role in various tasks, especially in syntactic parsing and language modeling. PCFGs are used to model the hierarchical structure of sentences and assign probabilities to different syntactic structures based on the likelihood of generating or parsing sentences.

**Inside NLP, PCFGs are used in the following ways:**

1. Syntactic Parsing: PCFGs are commonly used for parsing sentences and generating parse trees. Given a sentence, the PCFG aims to find the most probable parse tree by selecting the most likely production rules at each step of parsing. These parse trees represent the hierarchical syntactic structure of the sentences, which can be useful for various downstream tasks like information extraction, question answering, and sentiment analysis.
2. Ambiguity Resolution: Natural language often contains ambiguity, where a sentence can have multiple valid interpretations or parse trees. PCFGs help in disambiguating sentences by selecting the most probable parse tree based on the assigned probabilities of the production rules. This is particularly useful in tasks like machine translation, where choosing the correct translation can depend on the syntactic structure.
3. Language Modeling: PCFGs can be used for language modeling, where they estimate the probabilities of sentences or sequences of words. By assigning probabilities to different production rules, PCFGs can calculate the likelihood of generating a particular sentence according to the grammar, which aids in generating coherent and fluent sentences.
4. Speech Recognition: PCFGs have been used in certain approaches to speech recognition. They can help in modeling the syntactic structure of spoken sentences and aid in the conversion of spoken language into written text.
5. Grammar Induction: PCFGs can be used to induce grammars from a set of sentences. This is helpful in cases where the grammar of a language is not known beforehand or when dealing with languages with limited resources.

It's worth noting that while PCFGs have been widely used in NLP, more advanced probabilistic models, such as probabilistic dependency grammars and probabilistic phrase structure grammars, have been developed to address some of the limitations of PCFGs and achieve better accuracy and performance in various NLP tasks. Nevertheless, PCFGs remain an essential foundational concept in the field of computational linguistics and natural language processing.

**Outside Probabilities:**

In Natural Language Processing (NLP), the concept of "outside probabilities" is often associated with parsing algorithms, specifically those used in probabilistic context-free grammars (PCFGs) or more generally, in context-free grammars (CFGs) augmented with probabilities.

The "inside probabilities" refer to the probabilities associated with partial parses (subtrees) that span a contiguous subsequence of the input sentence. On the other hand, the "outside probabilities" are used to compute the probabilities of partial parses that cover the remaining portions of the input sentence, i.e., the portions that are not covered by the inside probabilities.

To explain it further, let's consider a parse tree of a sentence. The inside probabilities are used to calculate the probabilities of subtrees starting from the leaves of the tree and moving upwards towards the root. However, to get the probability of the entire sentence (the root of the tree), we need additional information from the outside probabilities.

In a PCFG or probabilistic CFG, the inside probabilities are usually calculated using algorithms like the Inside-Outside algorithm or the CYK algorithm (Cocke-Younger-Kasami). Once the inside probabilities have been computed, the outside probabilities can be calculated using a similar bottom-up approach, but this time starting from the root and moving towards the leaves of the tree.

**Outside probabilities are useful in various NLP tasks, including:**

1. Parsing Accuracy: When parsing sentences, both inside and outside probabilities are crucial for improving the accuracy of the parse. The combination of inside and outside probabilities allows for better disambiguation of sentence structures and more accurate parsing results.
2. Probabilistic Parsing: PCFGs can be used to compute probabilities for different parse trees of a sentence. By incorporating outside probabilities, the overall probabilities of the parse trees can be computed more accurately.
3. Parameter Estimation: In certain learning algorithms used for PCFGs, such as the Expectation-Maximization (EM) algorithm, outside probabilities are used to update the model parameters based on the likelihood of the training data.
4. Parsing with Unannotated Data: Outside probabilities are also employed in semi-supervised or unsupervised parsing, where the goal is to parse sentences using a combination of annotated and unannotated data.

It's important to note that while outside probabilities are beneficial for improving parsing accuracy and probabilistic modeling in NLP, they also increase the computational complexity of parsing algorithms. Hence, various approximation techniques and optimization methods are used to make the parsing process more efficient and scalable.

**Inside Outside Probabilities:**

In Natural Language Processing (NLP), "Inside-Outside" (IO) probabilities, also known as "Forward-Backward" probabilities, are used in the context of probabilistic context-free grammars (PCFGs) and probabilistic models to estimate probabilities of parse trees and compute expectations of different structures in the data.

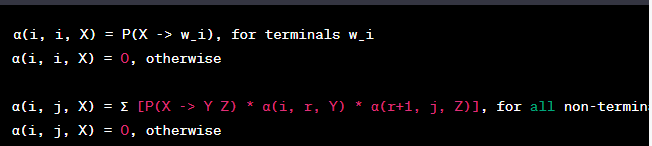
The Inside-Outside algorithm is a fundamental technique for calculating these probabilities. It efficiently computes two sets of probabilities: inside probabilities and outside probabilities.

**Inside Probabilities:**

1. Inside probabilities are used to compute the probability of a substructure (partial parse) of a sentence that spans a contiguous subsequence of the input. In the context of parsing, this refers to calculating the probability of a partial parse tree rooted at a particular node, given the observed words in the sentence up to that point. These probabilities are typically calculated in a bottom-up manner, starting from the leaves of the parse tree and moving towards the root. The inside probabilities are denoted as α(i, j, X), where "i" and "j" represent the span of the subsequence, and "X" is a non-terminal symbol.

The inside probabilities are calculated recursively using the following formula:

CSS

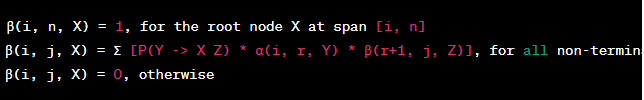


1. **Outside Probabilities:**

Outside probabilities are used to compute the probability of the remaining context of the sentence, which is not covered by the partial parse. They represent the probability of the unobserved words in the sentence outside the current span. The outside probabilities are denoted as β(i, j, X).

The outside probabilities are calculated recursively using the following formula:

CSS



Here, "n" represents the length of the input sentence.

**Applications of Inside-Outside probabilities in NLP include:**

1. Parsing: Inside-Outside probabilities are crucial for calculating the probabilities of parse trees and finding the most likely parse for a given sentence.
2. Parameter Estimation: Inside-Outside probabilities are used in the Expectation-Maximization (EM) algorithm for parameter estimation in PCFGs.
3. Probabilistic Modeling: Inside-Outside probabilities help estimate probabilities of different linguistic structures in the data, which is essential for various probabilistic models in NLP.

The Inside-Outside algorithm efficiently computes these probabilities and is widely used in various parsing and probabilistic modeling tasks in NLP.

**Dependency Grammars and Parsing Introduction:**

Dependency grammars and parsing are essential concepts in Natural Language Processing (NLP) that focus on analyzing the syntactic structure of sentences based on dependencies between words. Unlike phrase structure grammars, which use constituency-based parsing to represent hierarchical phrase structures, dependency grammars represent the relationships between individual words in a sentence using directed links called dependencies.

Here's an introduction to dependency grammars and parsing in NLP:

1. **Dependency Grammars:**

Dependency grammar is a type of formal grammar that focuses on the relationships between words in a sentence. In this approach, each word in the sentence is considered a node in a syntactic tree, and the links between the nodes represent the syntactic dependencies. The nodes are often labeled with the grammatical roles of the words, such as subject, object, modifier, etc. Dependency grammars aim to capture the grammatical relationships between words without explicit hierarchical structures like those found in phrase structure grammars.

1. **Dependency Parsing:**

Dependency parsing is the process of automatically analyzing the syntactic structure of a sentence using a dependency grammar. It involves creating a dependency tree (also known as a parse tree or a dependency graph) that represents the dependencies between words in the sentence. The goal of dependency parsing is to determine the grammatical relationships (dependencies) between words and to construct a tree that represents these relationships in a meaningful way.

1. **Dependency Relations:**

The links (arcs) in a dependency tree represent different types of dependency relations between words. Common dependency relations include:

* "nsubj": Nominal subject (the word that acts as the subject of the sentence)
* "dobj": Direct object (the word that is the direct object of the verb)
* "amod": Adjectival modifier (the word that modifies a noun)
* "advmod": Adverbial modifier (the word that modifies a verb or adjective)
* "conj": Conjunct (connecting words that have the same relationship to another word)
* "root": The root of the tree, usually representing the main verb of the sentence

1. Dependency Parsing Algorithms:

There are various algorithms used for dependency parsing, ranging from rule-based approaches to statistical and machine learning-based methods. Common dependency parsing algorithms include transition-based methods (e.g., arc-eager, arc-standard) and graph-based methods (e.g., Eisner's algorithm). These algorithms aim to find the most likely dependency tree for a given sentence based on the observed dependencies in a training corpus.

Dependency parsing is widely used in NLP for tasks such as information extraction, question answering, machine translation, and sentiment analysis. It provides valuable insights into the syntactic relationships between words in a sentence, which can be leveraged for a wide range of natural language understanding tasks.

**Transition Based Parsing:**

Transition-based parsing is a popular approach in Natural Language Processing (NLP) for syntactic parsing, particularly for dependency parsing. It involves using a sequence of transition actions to construct a dependency tree for a given sentence. The parser starts with an initial state and iteratively applies transition actions until it reaches a final state, representing the fully constructed dependency tree.

The primary components of transition-based parsing are:

**Configuration:**

A configuration represents the state of the parser at a particular step during parsing. It consists of a stack, a buffer, and a set of dependency arcs (links between words).

* Stack: The stack holds a sequence of partially processed words. Initially, it may contain only a special ROOT symbol representing the root of the tree.
* Buffer: The buffer contains the remaining words to be processed in the sentence.
* Dependency Arcs: The set of dependency arcs represents the dependencies that have been identified so far.

1. **Transition Actions:**

Transition actions are rules that define how the parser can change its configuration. Each action corresponds to an operation the parser can perform at a given state. Common transition actions include SHIFT (move a word from the buffer to the stack), LEFT-ARC (create a dependency arc from the top of the stack to the second-top word on the stack), and RIGHT-ARC (create a dependency arc from the second-top word on the stack to the top of the stack).

1. **Parsing Process:**

The parsing process starts with an initial configuration where the buffer contains all the words in the sentence, and the stack is empty except for the ROOT symbol. The parser then iteratively applies transition actions until the buffer is empty, and the stack contains only the ROOT symbol.

During each iteration, the parser uses a parsing model (e.g., a machine learning model) to predict the most appropriate transition action based on the current configuration and linguistic features of the words. The model is trained on annotated data (dependency treebanks) to learn the patterns and dependencies in the language.

1. **Deterministic or Non-Deterministic Parsing:**

Transition-based parsers can be either deterministic or non-deterministic. In deterministic parsers, the parser always chooses the highest-scoring action according to the model's predictions. Non-deterministic parsers may explore multiple possible actions, and some algorithms may use beam search to consider a limited set of promising actions.

Some common transition-based parsing algorithms include the arc-eager algorithm and the arc-standard algorithm. These algorithms differ in their transition actions and how they handle the construction of dependency arcs.

Transition-based parsing is known for its efficiency and simplicity, making it a popular choice for dependency parsing in various NLP applications. It has achieved impressive performance with the help of powerful machine learning models and feature representations.

**Formulation:**

In Natural Language Processing (NLP), "formulation" refers to the process of converting a natural language problem or task into a structured representation that can be processed by computational algorithms. Formulation involves defining the input and output formats, specifying the problem requirements, and deciding on the appropriate representation for the task at hand.

Formulation is a critical step in NLP as it lays the groundwork for developing algorithms, models, and systems to solve specific language-related tasks. The goal of formulation is to transform the unstructured and ambiguous nature of natural language into a more structured and well-defined format that can be handled by machines.

**Here are some examples of formulation in NLP:**

1. **Sentiment Analysis Formulation:**

Problem: Given a piece of text (e.g., a review or a tweet), determine the sentiment expressed in the text (e.g., positive, negative, neutral).

Formulation: In sentiment analysis, the formulation involves representing the text as a sequence of words (tokens) and mapping it to one of the sentiment classes (e.g., positive, negative, neutral). This can be achieved through supervised learning, where a labeled dataset of text examples with corresponding sentiments is used to train a machine learning model.

1. **Named Entity Recognition (NER) Formulation:**

Problem: Identify and classify named entities (e.g., person names, locations, organizations) in a given text.

Formulation: For NER, the formulation involves representing the text as a sequence of tokens and assigning a label to each token indicating whether it belongs to a named entity and its specific type (e.g., person, location, organization). This task can be addressed using sequence labeling approaches, such as Conditional Random Fields (CRFs) or deep learning models like BiLSTMs with CRF.

1. **Machine Translation Formulation:**

Problem: Translate a sentence or text from one language to another.

Formulation: In machine translation, the formulation requires mapping a source sentence in the source language to a target sentence in the target language. This involves representing the sentences as sequences of words and finding an appropriate mapping using various translation models, such as statistical machine translation or neural machine translation.

1. **Question Answering Formulation:**

Problem: Given a question in natural language, find the most relevant answer in a given context or document.

Formulation: For question answering, the formulation involves representing the question and the context/document as structured data. This can be achieved by using techniques like word embeddings, attention mechanisms, and language modeling to align the question with relevant parts of the context and generate an appropriate answer.

In each of these examples, the process of formulation helps define the task, establish the data representation, and guide the selection of appropriate algorithms and models to address the NLP problem effectively.

**Transition Based Parsing: Learning:**

Transition-based parsing in NLP often involves learning models that can predict the next transition action based on the current state of the parsing configuration. These models are typically machine learning models that are trained on annotated data (dependency treebanks) to learn the patterns and dependencies in the language. Learning in transition-based parsing consists of the following steps:

1. **Feature Extraction:**

To train a machine learning model for transition-based parsing, relevant features need to be extracted from the parsing configurations. Features can include information about the words in the stack and buffer, the current dependency arcs, and other linguistic features like part-of-speech tags and word embeddings. The goal is to create a feature representation that captures the relevant information necessary for predicting the next transition action.

1. **Training Data Preparation:**

Training data is essential for supervised learning of the parsing model. The training data consists of parsing configurations and the corresponding correct transition actions for a given set of sentences. These configurations can be obtained by applying a gold-standard oracle or an existing parser to generate the correct sequence of transitions for each sentence in the training set.

1. **Model Selection:**

Different machine learning models can be used for transition-based parsing, including linear models (e.g., logistic regression, linear SVM), decision trees, random forests, and neural network-based models (e.g., feedforward neural networks, recurrent neural networks, or transformers). The choice of the model depends on the complexity of the task and the availability of training data.

1. **Model Training:**

The training process involves feeding the extracted features and correct transition actions from the training data into the selected machine learning model. The model is then trained to learn the mapping between the feature representations and the correct transition actions. The objective is to minimize the prediction errors between the model's output and the gold-standard actions.

1. **Model Evaluation:**

After training, the model's performance is evaluated on a separate development or validation dataset. This dataset contains sentences that the model has not seen during training. The evaluation measures the accuracy of the model in predicting the correct transition actions. Metrics such as labeled attachment score (LAS) or unlabeled attachment score (UAS) are commonly used to evaluate dependency parsing accuracy.

1. **Hyperparameter Tuning:**

Machine learning models often have hyperparameters that need to be tuned to optimize the performance on the validation set. Hyperparameter tuning involves trying different combinations of hyperparameter values to find the best configuration for the model.

1. **Test Set Evaluation:**

Finally, the trained model is tested on a separate test set, which contains sentences that the model has never seen before. The test set evaluation provides a realistic assessment of the model's performance and its ability to generalize to new data.

By going through these steps, transition-based parsing models can be effectively trained and applied to parse sentences, providing valuable insights into the syntactic structure of natural language text.

**MST Based Dependency Parsing:**

MST (Minimum Spanning Tree) based dependency parsing is a popular approach in Natural Language Processing (NLP) for generating dependency trees from sentences. It involves finding the minimum spanning tree of a graph, where each word in the sentence is represented as a node, and the dependency relations between words are represented as weighted edges. The resulting minimum spanning tree corresponds to the dependency tree, where each word has exactly one head (governor) and a directed edge indicates the dependency relation.

Here's an overview of how MST-based dependency parsing works:

1. **Graph Construction:**

The first step in MST-based dependency parsing is to construct a graph representation of the sentence. Each word in the sentence is treated as a node, and the dependency relations between words are represented as weighted edges. These weights can be assigned based on various features, such as part-of-speech tags, word embeddings, or other linguistic properties. The goal is to create a graph that captures the likelihood of each dependency relation.

1. **Minimum Spanning Tree:**

The next step is to find the minimum spanning tree of the graph. The minimum spanning tree is a tree that connects all the nodes (words) in the graph with the minimum total edge weight while avoiding cycles. In dependency parsing, the minimum spanning tree corresponds to the dependency tree of the sentence, where each word has a single head (governor), and the edges indicate the dependency relations.

1. **Transition to Dependency Tree:**

Once the minimum spanning tree is obtained, the edges in the tree represent the dependency relations between words in the sentence. The parser then transforms this tree into a labeled dependency tree, where each edge is labeled with the corresponding dependency relation (e.g., subject, object, modifier).

1. **Non-Projective Dependency Parsing:**

MST-based dependency parsing can handle both projective and non-projective dependency structures. In a projective tree, the edges do not cross each other, and the tree can be drawn on a single line without crossing any edges. Non-projective trees, on the other hand, involve crossed edges and are more challenging to parse. MST-based approaches can handle non-projective structures efficiently.

1. **Dependency Parsing Algorithms:**

There are various algorithms for finding the minimum spanning tree of a graph, and they differ in terms of efficiency and optimality. Common algorithms used in MST-based dependency parsing include Chu-Liu/Edmonds' algorithm and Eisner's algorithm.

MST-based dependency parsing is widely used in NLP due to its efficiency and accuracy. It has been successfully applied to many languages and has become a standard method for parsing dependency structures. Machine learning techniques, such as structured prediction with structured perceptron or neural networks, are often integrated into MST-based parsing to improve performance further.

**MST-Based Dependency Parsing: Learning:**

MST-based dependency parsing can be combined with machine learning techniques to learn the weights of the edges in the graph representation of the sentence. By learning these weights from annotated data (dependency treebanks), the parsing model can capture the most probable dependency relations for different linguistic contexts. This approach is known as "MST-based dependency parsing with learning" or "graph-based dependency parsing with learning."

Here's how learning is incorporated into MST-based dependency parsing:

1. **Feature Extraction:**

Similar to transition-based parsing, feature extraction is a crucial step in MST-based dependency parsing with learning. Relevant features are extracted from the graph representation of the sentence, which includes information about the words (e.g., word embeddings, part-of-speech tags), the edges (e.g., the direction of the edge, the label of the edge), and other linguistic features that might be relevant for capturing the dependency relations.

1. **Training Data Preparation:**

Training data is essential for supervised learning of the parsing model. The training data consists of sentences along with their corresponding gold-standard dependency trees, where each dependency relation is represented as a labeled edge in the graph. For each sentence, the goal is to find the optimal set of edge weights that results in the correct dependency tree.

1. **Model Selection:**

Machine learning models are selected to learn the edge weights for the graph representation. Common choices include structured prediction models such as the structured perceptron or structured SVM, which can directly optimize the global structure (the dependency tree) rather than considering individual edges independently.

1. **Model Training:**

The training process involves feeding the extracted features and the correct dependency trees from the training data into the selected machine learning model. The model is then trained to learn the optimal weights for the edges that best fit the gold-standard dependency trees.

1. **Model Evaluation:**

After training, the model's performance is evaluated on a separate development or validation dataset using metrics such as labeled attachment score (LAS) or unlabeled attachment score (UAS). Hyperparameter tuning may be performed to optimize the model's performance on the validation set.

1. **Test Set Evaluation:**

Finally, the trained model is tested on a separate test set to evaluate its performance on unseen data. The test set evaluation provides a realistic assessment of the model's ability to generalize to new sentences and produce accurate dependency trees.

By combining MST-based dependency parsing with learning, the model can learn to make informed decisions about the most likely dependency relations in different linguistic contexts, leading to more accurate and linguistically informed dependency parsing results. The learned model can be applied to parse sentences and extract meaningful dependency structures, which are useful for various NLP tasks, such as information extraction, sentiment analysis, and machine translation.

**QUESTIONS:**

1. Elaborate the Syntax CKY (Cocke-Kasami-Younger) with algorithm steps.
2. Explain the concept of PCFGs (Probabilistic Context-Free Grammars) in detail.
3. Discuss the differences between dependency parsing and constituency parsing in NLP.
4. Discuss the role of Dependency Relations in Dependency Grammars. Explain the different types of dependency relations, such as "nsubj," "dobj," and "amod," and provide examples of each.
5. Explain the key steps involved in MST (Minimum Spanning Tree) based Dependency Parsing in detail.

**Unit 4**

**Distributional Semantics-Introduction**

**Introduction**

**Semantics**

It  is the study of [reference](https://en.wikipedia.org/wiki/Reference), [meaning](https://en.wikipedia.org/wiki/Meaning_(philosophy)), or [truth](https://en.wikipedia.org/wiki/Truth). The term can be used to refer to subfields of several distinct disciplines, including [philosophy](https://en.wikipedia.org/wiki/Philosophy), [linguistics](https://en.wikipedia.org/wiki/Linguistics) and [computer science](https://en.wikipedia.org/wiki/Computer_science).

**Distributional semantics** is a research area that develops and studies theories and methods for quantifying and categorizing semantic similarities between linguistic items based on their distributional properties in large samples of language data. The basic idea of distributional semantics can be summed up in the so-called [distributional](https://en.wikipedia.org/wiki/Distributionalism) hypothesis: *linguistic items with similar distributions have similar meanings.*

The **distributional hypothesis** in [linguistics](https://en.wikipedia.org/wiki/Linguistics) is derived from the [semantic theory](https://en.wikipedia.org/w/index.php?title=Semantic_theory&action=edit&redlink=1) of language usage, i.e. words that are used and occur in the same [contexts](https://en.wikipedia.org/wiki/Context_(language_use)) tend to purport similar meanings.

The underlying idea that "a word is characterized by the company it keeps" was popularized by [Firth](https://en.wikipedia.org/wiki/J._R._Firth) in the 1950s.

The distributional hypothesis is the basis for [statistical semantics](https://en.wikipedia.org/wiki/Statistical_semantics). Although the Distributional Hypothesis originated in linguistics, it is now receiving attention in [cognitive science](https://en.wikipedia.org/wiki/Cognitive_science) especially regarding the context of word use.

In recent years, the distributional hypothesis has provided the basis for the theory of [similarity-based generalization](https://en.wikipedia.org/w/index.php?title=Similarity-based_generalization&action=edit&redlink=1) in language learning: the idea that children can figure out how to use words they've rarely encountered before by generalizing about their use from distributions of similar words.

The distributional hypothesis suggests that the more semantically similar two words are, the more distributionally similar they will be in turn, and thus the more that they will tend to occur in similar linguistic contexts.

Distributional semantics is a framework in natural language processing and computational linguistics that focuses on representing the meaning of words based on their patterns of co-occurrence in large textual corpora. The underlying idea is that words that appear in similar contexts are likely to have similar meanings.

In distributional semantics, words are typically represented as high-dimensional vectors in a mathematical space, often referred to as a "semantic space" or "vector space." The vectors are derived from the statistical analysis of word co-occurrence patterns within a given corpus of text. This analysis can involve techniques like counting the frequency of words appearing together in sentences or using more advanced methods like neural networks to capture contextual relationships.

The key assumption in distributional semantics is the distributional hypothesis, which posits that words with similar distributions (i.e., similar patterns of co-occurrence with other words) tend to have similar meanings. This approach has been shown to capture various aspects of word meaning, including synonymy, antonymy, and even subtle semantic relationships.

Distributional semantics has been used in a variety of NLP tasks, including word similarity measurement, sentiment analysis, and even machine translation. However, it's worth noting that while distributional semantics is powerful, it does have limitations. For instance, it might struggle with capturing highly abstract or nuanced meanings, and it might not work well for words that have limited co-occurrence data in the training corpus.

## Distributional semantic modelling in vector spaces

Distributional semantics Favor the use of linear algebra as a computational tool and representational framework. The basic approach is to collect distributional information in high-dimensional vectors, and to define distributional/semantic similarity in terms of vector similarity.

Different kinds of similarities can be extracted depending on which type of distributional information is used to collect the vectors: **topical** similarities can be extracted by populating the vectors with information on which text regions the linguistic items occur in; **paradigmatic** similarities can be extracted by populating the vectors with information on which other linguistic items the items co-occur with. Note that the latter type of vectors can also be used to extract **syntagmatic** similarities by looking at the individual vector components.

The basic idea of a correlation between distributional and semantic similarity can be operationalized in many different ways.

**Distributional Models of Semantics**

Distributional semantic models have become a mainstay in NLP, providing useful features for downstream tasks. However, assessing long-term progress requires explicit long-term goals. In this paper, I take a broad linguistic perspective, looking at how well current models can deal with various semantic challenges. Given stark differences between models proposed in different subfields, a broad perspective is needed to see how we could integrate them. I conclude that, while linguistic insights can guide the design of model architectures, future progress will require balancing the often conflicting demands of linguistic expressiveness and computational tractability.

**Distributional Semantics: Applications:**

, Distributional semantics is a field of natural language processing (NLP) that focuses on representing the meaning of words and phrases based on their distributional patterns in large corpora of text. This approach has found numerous applications in various NLP tasks and beyond. Here are some key applications of distributional semantics:

1. **Word Embeddings**: Distributional semantics has led to the development of word embeddings like Word2Vec, GloVe, and FastText. These embeddings represent words as high-dimensional vectors, capturing semantic relationships between words. Word embeddings have been widely used in NLP tasks such as sentiment analysis, machine translation, and information retrieval.
2. **Semantic Similarity**: Distributional semantics enables the measurement of semantic similarity between words or phrases. This is useful in applications like query expansion in search engines, recommendation systems, and clustering of similar documents or entities.
3. **Sentiment Analysis**: Understanding the sentiment of text is crucial in applications like social media monitoring, product reviews, and customer feedback analysis. Distributional semantics can be used to capture the sentiment of words and phrases, helping to classify text as positive, negative, or neutral.
4. **Named Entity Recognition (NER):** Distributional semantics can aid in identifying named entities in text by leveraging contextual information. This is valuable in information extraction, chatbots, and question-answering systems.
5. **Topic Modeling:** Distributional representations can be employed to discover topics within a collection of documents. Latent Dirichlet Allocation (LDA), for example, uses distributional information to identify underlying topics in a corpus, making it useful in content recommendation and document categorization.
6. **Machine Translation**: Distributional semantics can improve the quality of machine translation by capturing word and phrase similarities across languages. This approach can help in handling polysemy and idiomatic expressions.
7. **Word Sense Disambiguation**: Distributional information can aid in resolving word sense ambiguities in natural language. It helps systems decide which meaning of a word is most appropriate based on the context in which it appears.
8. **Semantic Role Labeling**: Distributional representations can assist in identifying the roles that words play in a sentence, such as subject, object, or modifier. This is important in natural language understanding and question answering.
9. **Semantic Search**: Distributional semantics can enhance search engines by enabling more accurate retrieval of documents or information based on the semantic content of queries and documents.
10. **Question Answering**: Distributional semantics can be applied to question answering systems to understand the meaning of questions and retrieve relevant answers from large text corpora.
11. **Paraphrase Detection**: Detecting paraphrases (sentences or phrases with similar meanings) is crucial in applications like duplicate content detection, machine translation evaluation, and chatbot responses. Distributional semantics can help identify paraphrases by comparing the distributional patterns of words and phrases.
12. **Recommendation Systems**: In recommendation systems, distributional semantics can be used to understand user preferences and item characteristics, facilitating better recommendations in e-commerce, content recommendation, and personalized advertising.
13. **Text Summarization**: Distributional information can be used to identify important sentences or phrases in a document, aiding in automatic text summarization.
14. **Speech Recognition**: Distributional representations can improve automatic speech recognition systems by capturing the semantics of spoken language.

Distributional semantics has a wide range of applications in NLP and continues to be an active area of research, contributing to the development of more advanced and accurate natural language processing systems.

**Structured Models**

Query expansion is one of the most common methods to solve mismatch. We use the automatic term mismatch diagnosis to guide query expansion. Other forms of intervention, e.g. term removal or substitution, can also solve certain cases of mismatch, but they are not the focus of this work. We show that proper diagnosis can save expansion effort by 33%, while achieving near optimal performance. We generate structured expansion queries of Boolean conjunctive normal form (CNF) -- a conjunction of disjunctions where each disjunction typically contains a query term and its synonyms. Carefully created CNF queries are highly effective. They can limit the effects of the expansion terms to their corresponding query term, so that while fixing the mismatched terms, the expansion query is still faithful to the semantics of the original query. We show that CNF expansion leads to more stable retrieval across different levels of expansion, minimizing problems such as topic drift even with skewed expansion of part of the query.

**Word Embedding’s**

It is an approach for representing words and documents. Word Embedding or Word Vector is a numeric vector input that represents a word in a lower-dimensional space. It allows words with similar meaning to have a similar representation. They can also approximate meaning. A word vector with 50 values can represent 50 unique features.

**Features:** Anything that relates words to one another. Eg: Age, Sports, Fitness, Employed etc. Each word vector has values corresponding to these features.

**Goal of Word Embeddings**

* To reduce dimensionality
* To use a word to predict the words around it
* Inter word semantics must be captured

How are Word Embeddings used?

* They are used as input to machine learning models.  
  Take the words —-> Give their numeric representation —-> Use in training or inference
* To represent or visualize any underlying patterns of usage in the corpus that was used to train them.

**Implementations of Word Embeddings:**

Word Embeddings are a method of extracting features out of text so that we can input those features into a machine learning model to work with text data. They try to preserve syntactical and semantic information. The methods such as Bag of Words(BOW), CountVectorizer and TFIDF rely on the word count in a sentence but do not save any syntactical or semantic information. In these algorithms, the size of the vector is the number of elements in the vocabulary. We can get a sparse matrix if most of the elements are zero. Large input vectors will mean a huge number of weights which will result in high computation required for training. Word Embeddings give a solution to these problems.

Let’s take an example to understand how word vector is generated by taking emoticons which are most frequently used in certain conditions and transform each emoji into a vector and the conditions will be our features.

|  |  |  |  |
| --- | --- | --- | --- |
| **Happy** | ???? | ???? | ???? |
| **Sad** | ???? | ???? | ???? |
| **Excited** | ???? | ???? | ???? |
| **Sick** | ???? | ???? | ???? |

The emoji vectors for the emojis will be:

[happy,sad,excited,sick]

???? =[1,0,1,0]

???? =[0,1,0,1]

???? =[0,0,1,1]

.....

In a similar way, we can create word vectors for different words as well on the basis of given features. The words with similar vectors are most likely to have the same meaning or are used to convey the same sentiment.

**1) Word2Vec:**

In Word2Vec every word is assigned a vector. We start with either a random vector or **one-hot vector**.

**One-Hot vector:** A representation where only one bit in a vector is 1.If there are 500 words in the corpus then the vector length will be 500. After assigning vectors to each word we take a window size and iterate through the entire corpus. While we do this there are two**neural embedding methods** which are used:

**1.1) Continuous Bowl of Words(CBOW)**

In this model what we do is we try to fit the neighboring words in the window to the central word.

A screenshot of a computer screen

Description automatically generated

**1.2) Skip Gram**

In this model, we try to make the central word closer to the neighboring words. It is the complete opposite of the CBOW model. It is shown that this method produces more meaningful embeddings.

A screenshot of a computer

Description automatically generated

After applying the above neural embedding methods we get trained vectors of each word after many iterations through the corpus. These trained vectors preserve syntactical or semantic information and are converted to lower dimensions. The vectors with similar meaning or semantic information are placed close to each other in space.

**2) GloVe:**

This is another method for creating word embeddings. In this method, we take the corpus and iterate through it and get the co-occurrence of each word with other words in the corpus. We get a co-occurrence matrix through this. The words which occur next to each other get a value of 1, if they are one word apart then 1/2, if two words apart then 1/3 and so on.

Let us take an example to understand how the matrix is created. We have a small corpus:

Corpus:

It is a nice evening.

Good Evening!

Is it a nice evening?

|  | **it** | **is** | **a** | **nice** | **evening** | **good** |
| --- | --- | --- | --- | --- | --- | --- |
| **it** | 0 |  |  |  |  |  |
| **is** | 1+1 | 0 |  |  |  |  |
| **a** | 1/2+1 | 1+1/2 | 0 |  |  |  |
| **nice** | 1/3+1/2 | 1/2+1/3 | 1+1 | 0 |  |  |
| **evening** | 1/4+1/3 | 1/3+1/4 | 1/2+1/2 | 1+1 | 0 |  |
| **good** | 0 | 0 | 0 | 0 | 1 | 0 |

The upper half of the matrix will be a reflection of the lower half. We can consider a window frame as well to calculate the co-occurrences by shifting the frame till the end of the corpus. This helps gather information about the context in which the word is used.

Initially, the vectors for each word is assigned randomly. Then we take two pairs of vectors and see how close they are to each other in space. If they occur together more often or have a higher value in the co-occurrence matrix and are far apart in space then they are brought close to each other. If they are close to each other but are rarely or not frequently used together then they are moved further apart in space.

After many iterations of the above process, we’ll get a vector space representation that approximates the information from the co-occurrence matrix. The performance of GloVe is better than Word2Vec in terms of both semantic and syntactic capturing.

**Pre-trained Word Embedding Models:**

People generally use pre-trained models for word embeddings. Few of them are:

* SpaCy
* fastText
* Flair etc.

**Common Errors made:**

* You need to use the exact same pipeline during deploying your model as were used to create the training data for the word embedding. If you use a different tokenizer or different method of handling white space, punctuation etc. you might end up with incompatible inputs.
* Words in your input that doesn’t have a pre-trained vector. Such words are known as **Out of Vocabulary Word(oov). What** you can do is replace those words with “UNK” which means unknown and then handle them separately.
* Dimension mis-match: Vectors can be of many lengths. If you train a model with vectors of length say 400 and then try to apply vectors of length 1000 at inference time, you will run into errors. So make sure to use the same dimensions throughout.

**Benefits of using Word Embeddings:**

* It is much faster to train than hand build models like WordNet(which uses ***graph embeddings***)
* Almost all modern NLP applications start with an embedding layer
* It Stores an approximation of meaning

**Drawbacks of Word Embeddings:**

* It can be memory intensive
* It is corpus dependent. Any underlying bias will have an effect on your model
* It cannot distinguish between homophones. Eg: brake/break, cell/sell, weather/whether etc.

**Lexical Semantics**

The purpose of semantic analysis is to draw exact meaning, or you can say dictionary meaning from the text. The work of semantic analyzer is to check the text for meaningfulness.

We already know that lexical analysis also deals with the meaning of the words, then how is semantic analysis different from lexical analysis? Lexical analysis is based on smaller token but on the other side semantic analysis focuses on larger chunks. That is why semantic analysis can be divided into the following two parts −

### Studying meaning of individual word

It is the first part of the semantic analysis in which the study of the meaning of individual words is performed. This part is called lexical semantics.

### Studying the combination of individual words

In the second part, the individual words will be combined to provide meaning in sentences.

The most important task of semantic analysis is to get the proper meaning of the sentence. For example, analyze the sentence **“Ram is great.”** In this sentence, the speaker is talking either about Lord Ram or about a person whose name is Ram. That is why the job, to get the proper meaning of the sentence, of semantic analyzer is important.

## Elements of Semantic Analysis

Followings are some important elements of semantic analysis −

### Hyponymy

It may be defined as the relationship between a generic term and instances of that generic term. Here the generic term is called hypernym and its instances are called hyponyms. For example, the word color is hypernym and the color blue, yellow etc. are hyponyms.

### Homonymy

It may be defined as the words having same spelling or same form but having different and unrelated meaning. For example, the word “Bat” is a homonymy word because bat can be an implement to hit a ball or bat is a nocturnal flying mammal also.

### Polysemy

Polysemy is a Greek word, which means “many signs”. It is a word or phrase with different but related sense. In other words, we can say that polysemy has the same spelling but different and related meaning. For example, the word “bank” is a polysemy word having the following meanings −

* A financial institution.
* The building in which such an institution is located.
* A synonym for “to rely on”.

## Difference between Polysemy and Homonymy

Both polysemy and homonymy words have the same syntax or spelling. The main difference between them is that in polysemy, the meanings of the words are related but in homonymy, the meanings of the words are not related. For example, if we talk about the same word “Bank”, we can write the meaning ‘a financial institution’ or ‘a river bank’. In that case it would be the example of homonym because the meanings are unrelated to each other.

### Synonymy

It is the relation between two lexical items having different forms but expressing the same or a close meaning. Examples are ‘author/writer’, ‘fate/destiny’.

### Antonymy

It is the relation between two lexical items having symmetry between their semantic components relative to an axis. The scope of antonymy is as follows −

* **Application of property or not** − Example is ‘life/death’, ‘certitude/incertitude’
* **Application of scalable property** − Example is ‘rich/poor’, ‘hot/cold’
* **Application of a usage** − Example is ‘father/son’, ‘moon/sun’.

## Meaning Representation

Semantic analysis creates a representation of the meaning of a sentence. But before getting into the concept and approaches related to meaning representation, we need to understand the building blocks of semantic system.

### Building Blocks of Semantic System

In word representation or representation of the meaning of the words, the following building blocks play an important role −

* **Entities** − It represents the individual such as a particular person, location etc. For example, Haryana. India, Ram all are entities.
* **Concepts** − It represents the general category of the individuals such as a person, city, etc.
* **Relations** − It represents the relationship between entities and concept. For example, Ram is a person.
* **Predicates** − It represents the verb structures. For example, semantic roles and case grammar are the examples of predicates.

Now, we can understand that meaning representation shows how to put together the building blocks of semantic systems. In other words, it shows how to put together entities, concepts, relation and predicates to describe a situation. It also enables the reasoning about the semantic world.

## Approaches to Meaning Representations

Semantic analysis uses the following approaches for the representation of meaning −

* First order predicate logic (FOPL)
* Semantic Nets
* Frames
* Conceptual dependency (CD)
* Rule-based architecture
* Case Grammar
* Conceptual Graphs

## Need of Meaning Representations

A question that arises here is why do we need meaning representation? Followings are the reasons for the same −

### Linking of linguistic elements to non-linguistic elements

The very first reason is that with the help of meaning representation the linking of linguistic elements to the non-linguistic elements can be done.

### Representing variety at lexical level

With the help of meaning representation, unambiguous, canonical forms can be represented at the lexical level.

### Can be used for reasoning

Meaning representation can be used to reason for verifying what is true in the world as well as to infer the knowledge from the semantic representation.

## Lexical Semantics

The first part of semantic analysis, studying the meaning of individual words is called lexical semantics. It includes words, sub-words, affixes (sub-units), compound words and phrases also. All the words, sub-words, etc. are collectively called lexical items. In other words, we can say that lexical semantics is the relationship between lexical items, meaning of sentences and syntax of sentence.

Following are the steps involved in lexical semantics −

* Classification of lexical items like words, sub-words, affixes, etc. is performed in lexical semantics.
* Decomposition of lexical items like words, sub-words, affixes, etc. is performed in lexical semantics.
* Differences as well as similarities between various lexical semantic structures is also analyzed.

We understand that words have different meanings based on the context of its usage in the sentence. If we talk about human languages, then they are ambiguous too because many words can be interpreted in multiple ways depending upon the context of their occurrence.

Word sense disambiguation, in natural language processing (NLP), may be defined as the ability to determine which meaning of word is activated by the use of word in a particular context. Lexical ambiguity, syntactic or semantic, is one of the very first problem that any NLP system faces. Part-of-speech (POS) taggers with high level of accuracy can solve Word’s syntactic ambiguity. On the other hand, the problem of resolving semantic ambiguity is called WSD (word sense disambiguation). Resolving semantic ambiguity is harder than resolving syntactic ambiguity.

For example, consider the two examples of the distinct sense that exist for the word ***“bass”*** −

* I can hear bass sound.
* He likes to eat grilled bass.

The occurrence of the word **bass** clearly denotes the distinct meaning. In first sentence, it means **frequency** and in second, it means **fish**. Hence, if it would be disambiguated by WSD then the correct meaning to the above sentences can be assigned as follows −

* I can hear bass/frequency sound.
* He likes to eat grilled bass/fish.

## Evaluation of WSD

The evaluation of WSD requires the following two inputs −

### A Dictionary

The very first input for evaluation of WSD is dictionary, which is used to specify the senses to be disambiguated.

### Test Corpus

Another input required by WSD is the high-annotated test corpus that has the target or correct-senses. The test corpora can be of two types &minsu;

* **Lexical sample** − This kind of corpora is used in the system, where it is required to disambiguate a small sample of words.
* **All-words** − This kind of corpora is used in the system, where it is expected to disambiguate all the words in a piece of running text.

## Approaches and Methods to Word Sense Disambiguation (WSD)

Approaches and methods to WSD are classified according to the source of knowledge used in word disambiguation.

Let us now see the four conventional methods to WSD −

### Dictionary-based or Knowledge-based Methods

As the name suggests, for disambiguation, these methods primarily rely on dictionaries, treasures and lexical knowledge base. They do not use corpora evidences for disambiguation. The Lesk method is the seminal dictionary-based method introduced by Michael Lesk in 1986. The Lesk definition, on which the Lesk algorithm is based is **“measure overlap between sense definitions for all words in context”**. However, in 2000, Kilgarriff and Rosensweig gave the simplified Lesk definition as **“measure overlap between sense definitions of word and current context”**, which further means identify the correct sense for one word at a time. Here the current context is the set of words in surrounding sentence or paragraph.

### Supervised Methods

For disambiguation, machine learning methods make use of sense-annotated corpora to train. These methods assume that the context can provide enough evidence on its own to disambiguate the sense. In these methods, the words knowledge and reasoning are deemed unnecessary. The context is represented as a set of “features” of the words. It includes the information about the surrounding words also. Support vector machine and memory-based learning are the most successful supervised learning approaches to WSD. These methods rely on substantial amount of manually sense-tagged corpora, which is very expensive to create.

### Semi-supervised Methods

Due to the lack of training corpus, most of the word sense disambiguation algorithms use semi-supervised learning methods. It is because semi-supervised methods use both labelled as well as unlabeled data. These methods require very small amount of annotated text and large amount of plain unannotated text. The technique that is used by semisupervised methods is bootstrapping from seed data.

### Unsupervised Methods

These methods assume that similar senses occur in similar context. That is why the senses can be induced from text by clustering word occurrences by using some measure of similarity of the context. This task is called word sense induction or discrimination. Unsupervised methods have great potential to overcome the knowledge acquisition bottleneck due to non-dependency on manual efforts.

## Applications of Word Sense Disambiguation (WSD)

Word sense disambiguation (WSD) is applied in almost every application of language technology.

Let us now see the scope of WSD −

### Machine Translation

Machine translation or MT is the most obvious application of WSD. In MT, Lexical choice for the words that have distinct translations for different senses, is done by WSD. The senses in MT are represented as words in the target language. Most of the machine translation systems do not use explicit WSD module.

### Information Retrieval (IR)

Information retrieval (IR) may be defined as a software program that deals with the organization, storage, retrieval and evaluation of information from document repositories particularly textual information. The system basically assists users in finding the information they required but it does not explicitly return the answers of the questions. WSD is used to resolve the ambiguities of the queries provided to IR system. As like MT, current IR systems do not explicitly use WSD module and they rely on the concept that user would type enough context in the query to only retrieve relevant documents.

### Text Mining and Information Extraction (IE)

In most of the applications, WSD is necessary to do accurate analysis of text. For example, WSD helps intelligent gathering system to do flagging of the correct words. For example, medical intelligent system might need flagging of “illegal drugs” rather than “medical drugs”

### Lexicography

WSD and lexicography can work together in loop because modern lexicography is corpusbased. With lexicography, WSD provides rough empirical sense groupings as well as statistically significant contextual indicators of sense.

## Difficulties in Word Sense Disambiguation (WSD)

Followings are some difficulties faced by word sense disambiguation (WSD) −

### Differences between dictionaries

The major problem of WSD is to decide the sense of the word because different senses can be very closely related. Even different dictionaries and thesauruses can provide different divisions of words into senses.

### Different algorithms for different applications

Another problem of WSD is that completely different algorithm might be needed for different applications. For example, in machine translation, it takes the form of target word selection; and in information retrieval, a sense inventory is not required.

### Inter-judge variance

Another problem of WSD is that WSD systems are generally tested by having their results on a task compared against the task of human beings. This is called the problem of interjudge variance.

### Word-sense discreteness

Another difficulty in WSD is that words cannot be easily divided into discrete submeanings.

**WordNet**

it is required to understand the intuition of words in different positions and hold the similarity between the words as well. WordNET is a lexical database of semantic relations between words in more than 200 languages

In the field of natural language processing, there are a variety of tasks such as automatic [text classification](https://analyticsindiamag.com/guide-to-text-classification-using-textcnn/), [sentiment analysis](https://analyticsindiamag.com/guide-to-sentiment-analysis-using-bert/), [text summarization](https://analyticsindiamag.com/here-are-top-five-text-summarization-tools-that-could-be-helpful/), etc. These tasks are partially based on the pattern of the sentence and the meaning of the words in a different context. The two different words may be similar with an amount of amplitude. For example, the words ‘jog’ and ‘run’, both of them are partially different and also partially similar to each other. To perform specific [NLP](https://analyticsindiamag.com/top-open-source-nlp-projects-on-github-with-most-stars-in-2021-links-included/)-based tasks, it is required to understand the intuition of words in different positions and hold the similarity between the words as well. Here WordNET comes to the picture which helps in solving the linguistic problems of the NLP models.

WordNET is a lexical database of semantic relations between words in more than 200 languages.

WordNET is a lexical database of words in more than 200 languages in which we have adjectives, adverbs, nouns, and verbs grouped differently into a set of cognitive synonyms, where each word in the database is expressing its distinct concept. The cognitive synonyms which are called synsets are presented in the database with lexical and semantic relations.

**The Distinction Between WordNET and Thesaurus**

Where thesaurus is helping us in finding the synonyms and antonyms of the words the WordNET is helping us to do more than that. WordNET interlinks the specific sense of the words wherein thesaurus links words by their meaning only. In the WordNET the words are semantically disambiguated if they are in close proximity to each other. Thesaurus provides a level to the words in the network if the words have similar meaning but in the case of WordNET, we get levels of words according to their semantic relations which is a better way of grouping the words.

**Structure of WordNET**

The below image is a basic structure of the WordNET. The main concept of the relationship between the words in the WordNETs network is that the words are synonyms like sad and unhappy, benefit and profit. These words show the same concept of using them in similar contexts by interchanging them. These types of words are grouped into synsets which are unordered sets. Where synsets are linked together if they are having even small conceptual relations. Every synset in the network has its own brief definition and many of them are illustrated with the example of how to use them in a sentence. That definition and example part makes WordNET different from other

A diagram of a computer system

Description automatically generated

In the below picture we can see the structure of any synset where we are having synonyms of benefit in the array of synsets with the definition and the example of usage of benefit word. This synset is related to another synset word, where the words benefit and profit have exactly the same meaning.

A diagram of a business

Description automatically generated

Here we can see the structure of the wordnet and also how the synsets under the networks are interlinked because of the conceptual relation between the words.

**Relations in the WordNET**

**Hyponym:**In linguistics, a word with a broad meaning constitutes a category into which words with more specific meanings fall; a superordinate. For example, the colour is a hypernym of red. Where Hyponymy shows the relationship between a hypernym and a specific instance of a hyponym. A hyponym is a word or phrase whose semantic field is more specific than its hypernym. The semantic field of a hypernym, also known as a superordinate.

A diagram of different colors

Description automatically generated

[Image source](https://upload.wikimedia.org/wikipedia/commons/thumb/b/b4/Hyponym_and_hypernym.svg/1024px-Hyponym_and_hypernym.svg.png)

The above image is an example of the relationship between hyponyms and hypernym.

The reason for explaining these terms here is because in WordNET the most frequent relationships between synsets are based on these hyponym and hypernym relations. These are very beneficial in linking words like(paper, piece of paper). Saying more specifically with an example from the above picture like purple and violet, in WordNET the category colour includes purple which in turn includes violet. The root node of the hierarchy is the last point for every noun. In violet is a kind of purple and purple is a kind of colour then violet is a kind colour this is the hyponymy relation between the words which is transitive.

**Meronymy**: The wordnet hold follows the meronymy relation which defines the whole relationship between the synset for example a bike has two wheels handle and petrol tank. These components of a bike are inherited from their subordinates: if a bike has two wheels then a sports bike has wheels as well. In linguistics, we basically use this kind of relationship for adverbs which basically represents the characteristic of the noun. So the parts are inherited into a downward direction because all the bikes and types of bikes have two wheels, but not all kinds of automobiles consist of two wheels.

**Troponymy**: In linguistics, troponymy is the presence of a ‘manner’ relation between two lexemes. In WordNET  Verbs describing events that necessarily and unidirectionally entail one another are linked: {buy}-{pay}, {succeed}-{try}, {show}-{see}, etc. basically the in the hierarchy verbs towards the bottom shows the manners are characterizing the events like communication-talk-whisper.

**Antonymy**: Adjective words under the WordNET arranged in the antonymy pairs like wet and dry, smile and cry. Each of these pairs of antonyms is linked with sets of semantic similar ones. The cry is linked to weep, shed tears, sob, wail etc. so that they all can be considered as the opposite of indirect antonyms of a smile.

**Cross – PoS Relations**

Most of the relations in the wordNET are in the same [part of speech](https://analyticsindiamag.com/complete-tutorial-on-parts-of-speech-pos-tagging/). On the basis of part of speech relations, we can divide WordNET into 4 types of 4 subnets one for each noun, verbs, adjective, and adverb. There are also some cross-PoS pointers available in the network which include a morphosemantic link that holds the words with the same meaning and shares a stem. For example, many pairs like (reader read) in which the noun of the pair has a semantic layer with respect to the verb have been specified.

**Implementation of WordNET**

We can implement WordNET in just a few lines of code.

Importing libraries:

import nltk

from nltk.corpus import wordnet

Downloading the wordnet:

nltk.download('wordnet')

Output:

A black text on a white background

Description automatically generated

Taking trial of WordNET by checking the synonyms, antonyms and similarity percentage:

synonyms = [ ]

antonyms = [ ]

for synset in wordnet.synsets("evil"):

    for l in synset.lemmas( ):

        synonyms.append(l.name( ))

        if l.antonyms( ):

            antonyms.append(l.antonyms( )[0].name( ))

print(set(synonyms))

print(set(antonyms))

Output:



Here we can see the synonyms of the evil word and in the network, good and goodness is the opposite of the evil word.

Checking the word similarity feature:

word1 = wordnet.synset('man.n.01')

word2 = wordnet.synset('boy.n.01')

print(word1.wup\_similarity(word2)\*100)

Output:



Since we know grown-up boys are men, here when we asked the measure of similarity between the man and boy it gave the result around 66% which is a nice estimation of the similarity.

**Question Bank:**

**1.What is distributional semantics, and how does it work?**

**2. Explain the CBOW and Skip-Gram architectures in Word2Vec. When might one be more suitable than the other?**

**3.** **What are word embeddings, and how are they related to distributional semantics?**

**4.** **Explain the CBOW and Skip-Gram architectures in Word2Vec. When might one be more suitable than the other?**

**5.** **What is lexical semantics, and why is it important in natural language processing?**

**Unit 5**

**Text Summarization**

**Introduction:**

Text summarization is the process of automatically generating a concise and coherent summary of a longer text while preserving its most important information and overall meaning. This task is essential in information retrieval, document organization, and content consumption, as it allows users to quickly grasp the key points of a document without reading the entire text. There are two primary approaches to text summarization:

1. **Extractive Summarization**:

Extractive summarization involves selecting and extracting sentences or phrases directly from the source text to create a summary. It identifies the most relevant and informative segments of the original text and stitches them together to form a summary. Here's how extractive summarization works:

* + **Text Preprocessing**: The source text is preprocessed to remove stopwords, punctuation, and other noise. It may also be tokenized into sentences or phrases.
  + **Scoring Sentences**: Each sentence (or phrase) is assigned a relevance score based on various criteria such as word frequency, sentence length, and the presence of important keywords. Some advanced methods use machine learning models for scoring.
  + **Sentence Selection**: The sentences with the highest relevance scores are selected and arranged to create the summary. These selected sentences form the extractive summary.
  + **Output**: The extractive summary consists of sentences directly taken from the source text, arranged in a logical order.

Extractive summarization is relatively simpler to implement but may not always produce coherent summaries, as sentences are taken out of context. It's effective when the source text is well-structured and contains clear topic sentences.

1. **Abstractive Summarization**:

Abstractive summarization, on the other hand, generates summaries by paraphrasing and rewriting the source text in a more condensed form. Instead of extracting sentences verbatim, abstractive methods aim to understand the meaning of the text and generate human-like summaries. The process typically involves the following steps:

* + **Text Understanding**: The source text is processed to capture its main ideas, entities, and relationships. This may involve techniques like natural language understanding (NLU) and named entity recognition (NER).
  + **Content Representation**: A representation of the text's content is created, often using internal structures like semantic graphs or encoder-decoder models.
  + **Summary Generation**: A generative model, such as a neural network-based sequence-to-sequence model, is used to generate a summary based on the content representation. The model generates sentences that convey the key information in a coherent manner.
  + **Output**: The abstractive summary is a new text that may contain paraphrased content from the source text, expressed in a more concise and coherent form.

Abstractive summarization is more challenging but has the potential to produce summaries that are more human-like and contextually accurate. It is particularly useful for summarizing long and complex texts.

Text summarization has numerous applications, including:

* **News Summarization**: Generating concise summaries of news articles for quick consumption.
* **Content Curation**: Creating summaries of blog posts, research papers, or user-generated content to help readers decide what to read.
* **Legal Document Summarization**: Summarizing legal documents, contracts, and court cases to aid lawyers and legal professionals.
* **Document Indexing**: Creating summaries for document indexing and retrieval in information retrieval systems.
* **Search Engine Snippets**: Generating brief descriptions (snippets) for search engine results.
* **Email Summarization**: Automatically summarizing lengthy emails for improved email management.
* **Chatbot Responses**: Providing concise responses in chatbots and virtual assistants.

Text summarization continues to be an active research area, with ongoing advancements in natural language processing and machine learning techniques leading to more sophisticated and context-aware summarization methods.

**Optimization based Approaches for Summarization &Evaluation**

In Natural Language Processing we have two different applications; text summarization and text classification.

we will focus on one very important approach that is using LEXRANK algorithm for summarization**.**

**Summary:**

A Summary is a text that is produced from one or more texts that contains a significant portion of the information in the original text and that is no longer than half of the original text.

**Text summarization:**

Text Summarization is the process of distilling the most important information from a source to produce an abridged version for a particular user or task. And it is the process of generating short, fluent, and most importantly accurate summary of a respectively longer text document. The main idea behind automatic text summarization is to be able to find a short subset of the most essential information from the entire set and present it in a human-readable format. As online textual data grows, automatic text summarization methods have the potential to be very helpful because more useful information can be read in a short time.

**Text summarization** is the problem of reducing the number of sentences and words of a document without changing its meaning. There are different techniques to extract information from raw text data and use it for a summarization model, overall, they can be categorized as **Extractive**and**Abstractive.**Extractive methods select the most important sentences within a text (without necessarilyunderstanding the meaning), therefore the result summary is just a subset of the full text. On the contrary, Abstractive models use advanced NLP (i.e. word embeddings) to understand the semantics of the text and generate a meaningful summary. Consequently, Abstractive techniques are much harder to train from scratch as they need a lot of parameters and data.

**Automatic Text Summarization**

**Goal of a Text Summarization System**

* To give an overview of the original document in a shorter period.

**Summarization Applications:**

* Outlines or abstracts of any document, news article etc.
* Summaries of email threads.
* Action items from a meeting.
* Simplifying text by compressing sentences.

**Why automatic text summarization?**

1. Summaries reduce reading time.
2. Whenresearching documents, summaries make the selection process easier.
3. Automatic summarization improves the effectiveness of indexing.
4. Automatic summarization algorithms are less biased than human summarization.
5. Personalized summaries are useful in question-answering systems as they provide personalized information.
6. Using automatic or semi-automatic summarization systems enables commercial abstract services to increase the number of text documents they are able to process.

Types of summarizations:

1) Based on input type

2) Based on the purpose

3) Based on output type

**A diagram of text summarization

Description automatically generated**

1. Based on input type:
2. *S*ingle Document*,*where the input length is short. Many of the early summarization systems dealt with single-document summarization.
3. Multi-Document, where the input can be arbitrarily long.
4. Based on the purpose:
5. Generic, where the model makes no assumptions about the domain or content of the text to be summarized and treats all inputs as homogeneous. The majority of the work that has been done revolves around generic summarization.
6. Domain-specific, where the model uses domain-specific knowledge to form a more accurate summary. For example, summarizing research papers of a specific domain, biomedical documents, etc.
7. Query-based, where the summary only contains information that answers natural language questions about the input text.
8. Based on output type:
9. Extractive, where important sentences are selected from the input text to form a summary. Most summarization approaches today are extractive in nature.
10. Abstractive, where the model forms its own phrases and sentences to offer a more coherent summary, like what a human would generate. This approach is definitely more appealing, but much more difficult than extractive summarization.

**Text Classification:**

**Introduction:**

Text classification, also known as text categorization, is a classical problem in natural language processing (NLP), which aims to assign labels or tags to textual units such as sentences, queries, paragraphs, and documents. It has a wide range of applications including question answering, spam detection, sentiment analysis, news categorization, user intent classification, content moderation, and so on. Text data can come from different sources, including web data, emails, chats, social media, tickets, insurance claims, user reviews, and questions and answers from customer services, to name a few. Text is an extremely rich source of information. But extracting insights from text can be challenging and time-consuming, due to its unstructured nature.

Text classification can be performed either through manual annotation or by automatic labeling. With the growing scale of text data in industrial applications, automatic text classification is becoming increasingly important. Approaches to automatic text classification can be grouped into two categories:

• Rule-based methods

• Machine learning (data-driven) based methods

• Rule-based methods- It classify text into different categories using a set of pre-defined rules, and require a deep domain knowledge. On the other hand, machine learning based approaches learn to classify text based on observations of data. Using pre-labeled examples as training data, a machine learning algorithm learns inherent associations between texts and their labels.

• Machine learning models have drawn lots of attention in recent years. Most classical machine learning based models follow the two-step procedure. In the first step, some hand-crafted features are extracted from the documents (or any other textual unit). In the second step, those features are fed to a classifier to make a prediction. Popular hand-crafted features include bag of words (BoW) and their extensions. Popular choices of classification algorithms include Naïve Bayes, support vector machines (SVM), hidden Markov model (HMM), gradient boosting trees, and random forests. The two-step approach has several limitations. For example, reliance on the handcrafted features requires tedious feature engineering and analysis to obtain good performance. In addition, the strong dependence on domain knowledge for designing features makes the method difficult to generalize to new tasks. Finally, these models cannot take full advantage of large amounts of training data because the features (or feature templates) are pre-defined.

It is the process of categorizing the text into a group of words. By using NLP, text classification can automatically analyse text and then assign a set of predefined tags or categories based on its context. NLP is used for sentiment analysis, topic detection, and language detection.

**Text Classification Methods:**

First, we discuss how do we use a Naïve Bayes Classifiers for text classification.

Now, before moving to the formula for Naive Bayes, it is important to know about Bayes’ theorem.

**Bayes’ Theorem**

The British mathematician Reverend Thomas Bayes, **Bayes**‘ theorem is a mathematical formula used to determine the conditional probability, which is the likelihood of an outcome occurring based on a previous outcome.

A math equation with a circle and a circle

Description automatically generated with medium confidence

Using this formula, we can find the probability of A when B has occurred.

Here,

A is the proposition;

B is the evidence;

P(A) is the prior probability of proposition;

P(B) is the prior probability of evidence;

P(A/B) is called the posterior and

P(B/A) is called the likelihood.

Hence,

P**osterior = (Likelihood)(Proposition in prior probability)**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Evidence Prior probability**

This formula assumes that the predictors or features are independent, and one’s presence does not affect another’s feature. Hence, it is called ‘naïve.’

## Example Displaying Naïve Bayes Classifier

We are taking an example of a better understanding of the topic.

**Problem Statement:**

We are creating a classifier that depicts if a text is about sports or not.

The training data has five sentences:

|  |  |
| --- | --- |
| **Sentence** | **Label** |
| “A great game” | Sports |
| “The election was over” | Not sports |
| “Very clean match” | Sports |
| “It was a close election” | Not sports |
| “A clean but forgettable game” | Sports |

Here, you need to find the sentence ‘A very close game’ is of which label?

[Naive Bayes](https://www.upgrad.com/blog/naive-bayes-explained/), as a classifier, calculates the probability of the sentence “A very close game” is Sports with the probability ‘*Not Sports.’*

Mathematically, we want to know P (Sports | a very close game), probability of the label *Sports* in the sentence “A very close game.”

## Applying Bayes’ Theorem

We will convert the probability to be calculated using the count of the frequency of words. For this, we will use Bayes’ Theorem and some basic concepts of probability.

***P(A/B) = P(B/A) x P(A)***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***P(B)***

We have P (Sports | a very close game), and by using Bayes theorem, we will countermand the conditional probability:

***P (sports/ a very close game) = P(a very close game/ sports) x P(sports)***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***P (a very close game)***

We will abandon the divisor same for both the labels and compare

***P(a very close game/ Sports) x P(Sports)***

With

***P(a very close game/ Not Sports) x P(Not Sports)***

We can calculate the probabilities by calculating the counts the sentence *“A very close game”* emerges in the label ‘Sports’.  To determine P (a very close game | Sports), divide it by the total.

But, in the training data, ‘A very close game’ doesn’t seem anywhere so this probability is zero.

The model won’t be of much use without every sentence we want to classify is present in the training data.

## Naïve Bayes Classifier

Now comes the core part here, ‘*Naïve.’* Every word in a sentence is independent of the other, we’re not looking at the entire sentences, but at single words.

**P(a very close game) = P(a) x P(very) x P(close) x P(game)**

This presumption is powerful and useful too. The subsequent step is to apply:

***P(a very close game/Sports) = P(a/Sports) x P(very/Sports) x P(close/Sports) x P(game/Sports)***

These individual words appear many times in the training data that we can compute.

## Computing Probability

###### The finishing step is to calculate the probabilities and look at which one is larger.

###### First, we calculate the a priori probability of the labels: for the sentences in the given training data. The probability of it being Sports P (Sports) will be ⅗, and P (Not Sports) will be ⅖.

While calculating P (game/ Sports), we count the times the word “game” appears in *Sports* text (here 2) divided by the words in *sports* (11).

*P(game/Sports) = 2/11*

But, the word “close” isn’t present in any *Sports* text!

This means P (close | Sports) = 0 and is inconvenient as we will multiply it with other probabilities,

*P(a/Sports) x P(very/Sports) x 0 x P(game/Sports)*

The end result will be 0, and the entire calculation will be nullified. But this is not what we want, so we seek some other way around.

## Laplace Smoothing

We can eliminate the above issue with Laplace smoothing, where we will sum up 1 to every count; so that it is never zero.

We will add the possible number words to the divisor, and the division will not be more than 1.

In this case, the set of possible words are

*[‘a’, ‘great’, ‘very’, ‘over’, ‘it’, ‘but’, ‘game’, ‘match’, ‘clean’, ‘election’, ‘close’, ‘the’, ‘was’, ‘forgettable’].*

The possible number of words is 14; by applying Laplace smoothing,

*P(game/Sports) = 2+1*

 *\_\_\_\_\_\_\_\_\_\_\_*

  *11 + 14*

Final Outcome:

|  |  |  |
| --- | --- | --- |
| **Word** | **P (word | Sports)** | **P (word | Not Sports)** |
| a | (2 + 1) ÷ (11 + 14) | (1 + 1) ÷ (9 + 14) |
| very | (1 + 1) ÷ (11 + 14) | (0 + 1) ÷ (9 + 14) |
| close | (0 + 1) ÷ (11 + 14) | (1 + 1) ÷ (9 + 14) |
| game | (2 + 1) ÷ (11 + 14) | (0 + 1) ÷ (9 + 14) |

Now, multiplying all the probabilities to find which is bigger:

**P(a/Sports) x P(very/Sports) x P(game/Sports)x P(game/Sports)x P(Sports)**

**= 2.76 x 10 ^-5**

**= 0.0000276**

**P(a/Non Sports) x P(very/ Non Sports) x P(game/ Non Sports)x P(game/ Non Sports)x P(Non Sports)**

**= 0.572 x 10 ^-5**

**= 0.00000572**

Hence, we have finally got our classifier that gives “A very close game” the label Sports as its probability is high and we infer that the sentence belongs to the Sports category.

## Types of Naive Bayes Classifier

###### Now that we have understood what a Naïve Bayes Classifier is and have seen an example too, let’s see the types of it:

### 1. Multinomial Naive Bayes Classifier

This is used mostly for document classification problems, whether a document belongs to the categories such as politics, sports, technology, etc. The predictor used by this classifier is the frequency of the words in the document.

we introduce the multinomial naive Bayes classifier, so called because it is a Bayesian classifier that makes a simplifying (naive) assumption about how the features interact.

We represent a text document bag of words as if it were a bag of words, that is, an unordered set of words with their position ignored, keeping only their frequency in the document. instead of representing the word order in all the phrases like “I love this movie” and “I would recommend it”, we simply note that the word I occurred 5 times in the entire excerpt, the word it 6 times, the words love, recommend, and movie once, and so on.

Naive Bayes is a probabilistic classifier, meaning that for a document d, out of all classes c ∈ C the classifier returns the class ˆc which has the maximum posterior ˆ probability given the document. we use the hat notation ˆ to mean “our estimate of the correct class”.

cˆ = argmax c∈C P(c|d)………..(1)

This idea of Bayesian inference has been known since the work of Bayes (1763), Bayesian inference and was first applied to text classification by Mosteller and Wallace (1964). The intuition of Bayesian classification is to use Bayes’ rule to transform Eq. 4.1 into other probabilities that have some useful properties. Bayes’ rule is presented in Eq. 4.2; it gives us a way to break down any conditional probability P(x|y) into three other probabilities:

P(x|y) = P(y|x)P(x) / P(y) ………….(2)

We can then substitute Eq. 2 into Eq. 1 to get Eq. 3:

cˆ = argmax c∈C P(c|d) = argmax c∈C P(d|c)P(c) / P(d) ….(3) We can conveniently simplify Eq. 4.3 by dropping the denominator P(d). This is possible because we will be computing P(d|c)P(c)/P(d) for each possible class. But P(d) doesn’t change for each class; we are always asking about the most likely class for the same document d, which must have the same probability P(d). Thus, we can choose the class that maximizes this simpler formula:

cˆ = argmax c∈C P(c|d) = argmax c∈C P(d|c)P(c) ……(4)

We call Naive Bayes a generative model because we can read Eq. 4.4 as stating a kind of implicit assumption about how a document is generated: first a class is sampled from P(c), and then the words are generated by sampling from P(d|c). (In fact we could imagine generating artificial documents, or at least their word counts, by following this process).

we compute the most probable class ˆc given some document d by choosing the class which has the highest product of two probabilities: the prior probability of the class P(c) and the likelihood of the document P(d|c):

cˆ = argmax c∈C P(d|c) P(c) …. (5)

Without loss of generalization, we can represent a document d as a set of features f1, f2,..., fn:

cˆ = argmax c∈C P(f1, f2,...., fn|c) P(c) …….(6)

Unfortunately, Eq. 6 is still too hard to compute directly: without some simplifying assumptions, estimating the probability of every possible combination of features (for example, every possible set of words and positions) would require huge numbers of parameters and impossibly large training sets. Naive Bayes classifiers therefore make two simplifying assumptions.

The first is the bag-of-words assumption discussed intuitively above: we assume position doesn’t matter, and that the word “love” has the same effect on classification whether it occurs as the 1st, 20th, or last word in the document. Thus we assume that the features f1, f2,..., fn only encode word identity and not position.

The second is commonly called the naive Bayes assumption: this is the condi- naive Bayes assumption tional independence assumption that the probabilities P(fi |c) are independent given the class c and hence can be ‘naively’ multiplied as follows:

P(f1, f2,...., fn|c) = P(f1|c)·P(f2|c)· ... ·P(fn|c) ……(7)

The final equation for the class chosen by a naive Bayes classifier is thus:

cNB = argmax c∈C P(c) ℿ f∈F P(f |c) …..(8)

To apply the naive Bayes classifier to text, we need to consider word positions, by simply walking an index through every word position in the document:

positions ← all word positions in test document

cNB = argmax c∈C P(c) ℿ i∈positions P(wi |c) ……..(9)

Naive Bayes calculations, like calculations for language modeling, are done in log space, to avoid underflow and increase speed. Thus Eq. 9 is generally instead expressed as

cNB = argmax c∈C logP(c) +⅀ i∈positions logP(wi |c)….. (10)

By considering features in log space, Eq. 10 computes the predicted class as a linear function of input features. Classifiers that use a linear combination of the inputs to make a classification decision —like naive Bayes and also logistic regression are called linear classifiers.

### 2. Bernoulli Naive Bayes Classifier

This is similar to the multinomial **Naive Bayes Classifier,** but its predictors are boolean variables. The parameters we use to predict the class variable take up the values yes or no only. For instance, whether a word occurs in a text or not.

### 3. Gaussian Naive Bayes Classifier

When the predictors take a constant value, we assume that these values are sampled from a Gaussian distribution.

A black background with a black square

Description automatically generated with medium confidence

Naive Bayes is commonly used for text classification in applications such as predicting spam emails, classifying text (e. g. news) into categories such as politics, sports, lifestyle etc. In general, Naïve Bayes has proven to perform well in text classification application.

**Introduction of Sentiment Analysis**

Sentiment analysis is a NLP technique used to determine whether a given data is positive, negative, or neutral. Sentiment analysis is basically often performed on textual data to help businesses monitor brand and product sentiment in customer feedback and understanding customer needs.

We focus on one common text categorization task, sentiment analysis, the ex- sentiment analysis traction of sentiment, the positive or negative orientation that a writer expresses toward some object. A review of a movie, book, or product on the web expresses the author’s sentiment toward the product, while an editorial or political text expresses sentiment toward a candidate or political action. Extracting consumer or public sentiment is thus relevant for fields from marketing to politics. The simplest version of sentiment analysis is a binary classification task, and the words of the review provide excellent cues. Consider, for example, the following phrases extracted from positive and negative reviews of movies and restaurants. Words like great, richly, awesome, and pathetic, and awful and ridiculously are very informative cues:

+ ...zany characters and richly applied satire, and some great plot twists

− It was pathetic. The worst part about it was the boxing scenes...

+ ...awesome caramel sauce and sweet toasty almonds. I love this place!

− ...awful pizza and ridiculously overpriced...

Sentiment analysis aims to estimate the **sentiment polarity**of a body of text based solely on its content. The sentiment polarity of text can be defined as a value that says whether the expressed opinion is **positive** (polarity=**1**), **negative** (polarity=**0**), or neutral.

While standard naive Bayes text classification can work well for sentiment analysis, some small changes are generally employed that improve performance. First, for sentiment classification and several other text classification tasks, whether a word occurs or not seems to matter more than its frequency. Thus, it often improves performance to clip the word counts in each document at 1 (see the end of the chapter for pointers to these results). This variant is called binary multinomial naive Bayes or binary naive Bayes. The variant uses the same algorithm as binary naive Bayes except that for each document we remove all duplicate words before concatenating them into the single big document during training and we also remove duplicate words from test documents. Fig. 4.3 shows an example in which a set of four documents (shortened and text-normalized for this example) are remapped to binary, with the modified counts shown in the table on the right. The example is worked without add-1 smoothing to make the differences clearer. Note that the results counts need not be 1; the word great has a count of 2 even for binary naive Bayes, because it appears in multiple documents.

A second important addition commonly made when doing text classification for sentiment is to deal with negation. Consider the difference between I really like this movie (positive) and I didn’t like this movie (negative). The negation expressed by didn’t completely alters the inferences we draw from the predicate like. Similarly, negation can modify a negative word to produce a positive review (don’t dismiss this film, doesn’t let us get bored). A very simple baseline that is commonly used in sentiment analysis to deal with negation is the following: during text normalization, prepend the prefix NOT to every word after a token of logical negation (n’t, not, no, never) until the next punctuation mark. Thus the phrase

didn’t like this movie , but I

becomes

didn’t NOT\_like NOT\_this NOT movie , but I

Newly formed ‘words’ like NOT like, NOT recommend will thus occur more often in negative document and act as cues for negative sentiment, while words like NOT bored, NOT dismiss will acquire positive associations. We will return in Chapter 20 to the use of parsing to deal more accurately with the scope relationship between these negation words and the predicates they modify, but this simple baseline works quite well in practice.

Finally, in some situations we might have insufficient labelled training data to train accurate naive Bayes classifiers using all words in the training set to estimate positive and negative sentiment. In such cases we can instead derive the positive and negative word features from sentiment lexicons, lists of words that are pre- sentiment lexicons annotated with positive or negative sentiment.

Simple Sentiment Analysis Methods:

Sentiment analysis, also known as opinion mining, is the process of determining the sentiment or emotional tone expressed in a piece of text. Simple sentiment analysis methods are basic techniques for classifying text as positive, negative, or neutral in sentiment. Here are some straightforward approaches to sentiment analysis:

1. **Lexicon-Based Sentiment Analysis**:

Lexicon-based methods rely on sentiment lexicons or dictionaries that contain lists of words or phrases associated with different sentiment polarities (positive, negative, or neutral). Here's how you can perform lexicon-based sentiment analysis:

* + **Tokenization**: Split the text into individual words or tokens.
  + **Lexicon Lookup**: Check each word against the sentiment lexicon and assign a sentiment polarity (positive, negative, or neutral) based on the presence of words in the lexicon.
  + **Sentiment Aggregation**: Calculate a sentiment score for the entire text by summing or averaging the sentiment polarities of individual words.
  + **Thresholding**: Apply a threshold to the sentiment score to classify the text as positive, negative, or neutral.

One common lexicon-based approach is the VADER (Valence Aware Dictionary and sEntiment Reasoner) sentiment analysis tool, which is available as a Python library.

1. **Bag-of-Words (BoW) with Supervised Learning**:

This method uses a supervised machine learning model, such as logistic regression or Naive Bayes, to classify text based on the presence and frequency of words in a predefined vocabulary. Here's how it works:

* + **Feature Extraction**: Create a feature vector for each text document, where each feature represents the presence or frequency of a word from the predefined vocabulary (Bag-of-Words representation).
  + **Training**: Train a supervised machine learning model on a labeled dataset of text samples with sentiment labels (positive, negative, or neutral).
  + **Classification**: Use the trained model to predict the sentiment of new text samples.

You'll need a labeled dataset for training, where each text sample is associated with its corresponding sentiment label.

1. **TextBlob**:

TextBlob is a Python library that simplifies text processing tasks, including sentiment analysis. It provides a straightforward API for sentiment analysis with a pre-trained model. Here's how to perform sentiment analysis using TextBlob:

from textblob import TextBlob

text = "I love this product. It's amazing!"

analysis = TextBlob(text)

# Get sentiment polarity (positive, negative, or neutral)

sentiment = analysis.sentiment.polarity

if sentiment > 0:

print("Positive sentiment")

elif sentiment < 0:

print("Negative sentiment")

else:

print("Neutral sentiment"))

TextBlob's sentiment analysis is based on a simple rule-based approach.

1. **Naive Rule-Based Approaches**:

You can define your own simple rules or heuristics to determine sentiment based on specific keywords or patterns in the text. For example, if a text contains words like "good," "excellent," "happy," it can be classified as positive.

Simple sentiment analysis methods are a good starting point for basic sentiment classification tasks. However, they have limitations in handling nuances, sarcasm, context, and domain-specific language. For more accurate and robust sentiment analysis, especially in real-world applications, more advanced techniques such as deep learning-based models (e.g., LSTM, BERT) and fine-tuning on large sentiment analysis datasets are often used.

**Interview Questions:**

**1**.what is text summarization and why it is important in NLP?

### 2. **What is the Naive Bayes algorithm, and where is it used in NLP?**

### **3.**