DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
YEAR/SEM: III /VI
23CSX503, NATURAL LANGUAGE PROCESSING FUNDAMENTALS
UNIT I
INTRODUCTION
MODULE

Natural Language Processing tasks in syntax, semantics, and pragmatics - challenges of NLP - NLP Phases - Language Modeling: Grammar-based LM, Statistical LM - Regular Expressions, Finite-State Automata – English Morphology, Transducers for lexicon and rules, Tokenization, Detecting and Correcting Spelling Errors, Minimum Edit Distance.

1.1 INTRODUCTION

Natural Language Processing (NLP) is the sub-field of Computer Science especially Artificial Intelligence (AI) that is concerned about enabling computers to understand and process human language. Technically, the main task of NLP would be to program computers for analyzing and processing huge amount of natural language data.

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

1.1.1 History of NLP

The history of Natural Language Processing (NLP) is divided into four main phases based on focus and development style.

First Phase – Machine Translation (Late 1940s – Late 1960s)

- Focused mainly on machine translation (MT).
- Began with Weaver's memorandum in 1949.
- 1954 Georgetown-IBM experiment showed Russian-to-English translation.
- MT journal and international MT conferences started.
- Major progress seen in the 1961 Teddington conference.

Second Phase – AI Influenced (Late 1960s – Late 1970s)

- Emphasized world knowledge and meaning representation.
- Influenced by artificial intelligence.
- Development of knowledge bases and early question-answering systems like BASEBALL.
- Advanced systems introduced inference over knowledge bases.

Third Phase – Grammatico-Logical (Late 1970s – Late 1980s)

- Shifted towards logic-based representations.
- Developed sentence processors like Core Language Engine.
- Produced practical tools such as parsers and database query systems.
- Strong focus on lexicon development.

Fourth Phase – Lexical & Corpus (1990s)

- Introduced corpus-based and lexicalized grammar approaches.
- Major growth due to machine learning techniques in NLP.

Definition

Natural Language Processing (NLP) is a field of artificial intelligence (AI) that focuses on enabling computers to understand, interpret, and generate human language. NLP involves a range of techniques and methodologies to process and analyze text or speech data.

Basics of NLP:

1. **Tokenization:** Tokenization is the process of breaking text into smaller units called tokens, such as words or subwords. It serves as the first step in NLP and is crucial for subsequent analysis.
2. **Part-of-Speech (POS) Tagging:** POS tagging involves assigning grammatical tags to each word in a sentence, indicating its part of speech (e.g., noun, verb, adjective). It helps in understanding the syntactic structure and grammatical relationships within a sentence.
3. **Named Entity Recognition (NER):** NER aims to identify and classify named entities in text, such as person names, organization names, locations, dates, etc. It helps in extracting relevant information from text and understanding the context.
4. **Parsing and Syntactic Analysis:** Parsing involves analyzing the grammatical structure of sentences. Syntactic analysis helps in understanding the relationships between words and their hierarchical structure within a sentence.
5. **Sentiment Analysis:** Sentiment analysis, also known as opinion mining, involves determining the sentiment or emotional tone expressed in text. It can classify text as positive, negative, or neutral, helping in understanding public opinion, customer feedback analysis, and social media sentiment analysis.
6. **Machine Translation:** Machine translation focuses on automatically translating text from one language to another. It involves developing models and algorithms to understand the source language and generate coherent translations in the target language.
7. **Question Answering:** Question answering aims to provide accurate and relevant answers to user queries based on a given context or a corpus of knowledge. It involves understanding the questions, retrieving relevant information, and generating appropriate answers.
8. **Text Generation:** Text generation involves creating human-like text based on a given prompt or context. It can be used for chatbots, automated content creation, or generating personalized responses.
9. **Information Extraction:** Information extraction aims to identify and extract structured information from unstructured text. It involves tasks such as entity extraction, relation extraction, and event extraction.
10. **Text Classification:** Text classification involves categorizing text into predefined categories or topics. It can be used for sentiment analysis, spam detection, topic classification, and more.
11. **Summarization:** Text summarization focuses on generating concise and coherent summaries of longer texts. It can be extractive (selecting and combining important sentences) or abstractive (generating new sentences).
12. **Language Generation:** Language generation involves generating natural language text that is coherent and contextually appropriate. It includes tasks like text completion, dialogue generation, and storytelling.

1.1.2 COMPONENTS OF NLP

There are two components of NLP as given:

1. Natural Language Understanding (NLU) 2. Natural Language Generation (NLG)

1. Natural Language Understanding (NLU)

Understanding involves the following tasks −

- Mapping the given input in natural language into useful representations.
- Analyzing different aspects of the language.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

2. Natural Language Generation (NLG)

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

It involves −

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

1.2 NATURAL LANGUAGE PROCESSING TASKS IN SYNTAX, SEMANTICS, AND PRAGMATICS

In the field of Natural Language Processing (NLP), tasks are often categorised based on the level of linguistic knowledge required to resolve ambiguity and extract meaning. According to the sources, these levels include syntax, semantics, and pragmatics.

1. Natural Language Processing Tasks in Syntax

Syntax refers to the knowledge needed to order and group words together based on structural relationships. Key tasks in this area include:

- **Part-of-Speech (POS) Tagging:** This involves resolving lexical ambiguity by deciding whether a word serves as a specific category, such as a noun or a verb, in a given context.
- **Parsing:** This is the process of mapping a string of words to its constituent structure or parse tree using a formal grammar, such as a Context-Free Grammar (CFG). It explores the structural relationships between words, such as identifying the subject and object of a sentence.
- **Subcategorisation:** This task involves modelling the constraints that specific words, particularly verbs, place on the number and type of phrases that can accompany them (for example, distinguishing between transitive and intransitive verbs).
- **Dependency Parsing:** Unlike constituency-based parsing, this task describes syntactic structure in terms of binary relations between individual words rather than groups of words.

2. Natural Language Processing Tasks in Semantics

Semantics is the study of meaning, both at the level of individual words and how they combine to form the meaning of a sentence. Relevant tasks include:

- **Word Sense Disambiguation (WSD):** This is the process of determining which **word sense** is being used when a word has multiple potential meanings. It often relies on **lexical relations** (like synonymy or hyponymy) or **contextual similarity**.
- **Semantic Role Labelling (SRL):** This task identifies the **semantic roles** played by different entities in a sentence with respect to a predicate, such as determining who is the **Agent** (the causer) and who is the **Theme** (the participant affected).
- **Compositional Semantic Analysis:** This is the process of constructing a **meaning representation** for an entire sentence by combining the meanings of its smaller constituent parts.
- **Selectional Restriction Analysis:** This involves modelling the **semantic type constraints** that predicates impose on their arguments, such as the fact that the verb *eat* generally requires an **edible** object.

3. Natural Language Processing Tasks in Pragmatics and Discourse

Pragmatics and discourse involve the relationship between meaning and the goals and intentions of the speaker, as well as units of language larger than a single utterance. Tasks include:

- **Coreference Resolution:** This is the task of determining when multiple linguistic expressions (like a name and a pronoun) refer to the same real-world entity.
- **Speech Act Interpretation:** This involves deciding what kind of action a speaker intends to perform with an utterance, such as making a request, a statement, or asking a question.
- **Coherence Relation Assignment:** This task involves identifying the meaningful connections between utterances in a text, such as Explanation, Elaboration, or Contrast.
- **Dialogue Management:** In conversational agents, this task controls the structure of the dialogue, keeping track of the information state (context and goals) and deciding on the system's next action.
- **Grounding:** This is the process by which participants in a conversation establish mutual understanding or "common ground" by acknowledging or demonstrating that they have understood the previous utterance.

1.3 CHALLENGES OF NLP

1. Language Diversity

Human language is extremely diverse. There are thousands of languages spoken across the world, and each language has its own grammar rules, vocabulary, sentence structure, and cultural expressions. Even within the same language, there are different dialects and styles of speaking. NLP systems must be trained separately for each language, which makes language processing complex and resource-intensive.

2. Ambiguity in Natural Language

Natural language is often ambiguous, meaning the same word or sentence can have multiple meanings depending on the context. For example, the sentence *"I saw her duck"* can have different interpretations. NLP systems find it difficult to identify the correct meaning without deep contextual understanding, which affects accuracy.

3. Lack of High-Quality Training Data

NLP models require large volumes of clean, labeled, and high-quality text data for training. However, such data is not always available, especially for regional or low-resource languages. Preparing annotated datasets requires time, cost, and expert knowledge, which limits the performance of NLP systems.

4. Words with Multiple Meanings (Polysemy)

Many words in natural language have more than one meaning. The correct meaning depends on how the word is used in a sentence. For example, the word *"bat"* can refer to an animal or a sports tool. Identifying the correct meaning is challenging for NLP systems and requires effective word sense disambiguation techniques.

5. Spelling and Grammatical Errors

Text data created by humans often contains spelling mistakes, grammatical errors, abbreviations, and informal language. Social media text, chat messages, and reviews are highly unstructured. These errors introduce noise and reduce the accuracy of NLP models during text analysis.

6. Understanding Context

Understanding the full meaning of a sentence often requires information from surrounding text or previous conversation. For example, pronouns like *"he"*, *"she"*, or *"it"* depend on earlier references. NLP systems struggle to maintain long-term context, especially in conversational applications like chatbots.

7. Bias in NLP Models

NLP models learn patterns from training data that may contain social, cultural, or gender biases. As a result, models may produce biased or unfair outputs. This is a serious ethical challenge, especially

in applications such as hiring, customer service, and automated decision-making systems.

8. Multilingual and Cross-Language Processing

Supporting multiple languages is difficult because not all languages have enough digital resources or annotated datasets. Differences in syntax, word order, and grammar across languages further increase complexity. Machine translation between languages also introduces errors and loss of meaning.

9. High Computational and Resource Requirements

Modern NLP models, especially deep learning models, require large computational resources such as GPUs, high memory, and long training time. This increases development cost and makes NLP solutions less accessible for small organizations and researchers.

10. Maintaining Continuous and Natural Conversations

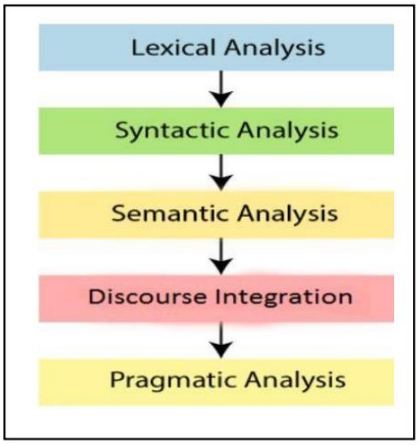
Conversational NLP systems must understand user intent, remember past interactions, and generate meaningful responses. Maintaining context over long conversations and responding naturally like a human is still a challenging task for NLP systems.

1.4 NLP PHASES

There are general five phases: 1. Lexical Analysis 2. Syntax Analysis (Parsing) 3. Semantic Analysis 4. Discourse Integration 5. Pragmatic Analysis

1. Lexical Analysis: (Morphological Processing)

- The first phase of NLP is the Lexical Analysis.
- The purpose of this phase is to break chunks of language input into sets of tokens corresponding to paragraphs, sentences and words.
- This phase scans the source code as a stream of characters and converts it into meaningful lexemes. It divides the whole text into paragraphs, sentences, and words.
- **Lexeme:** A lexeme is a basic unit of meaning. In linguistics, the abstract unit of morphological analysis that corresponds to a set of forms taken by a single word is called a lexeme. The way in which a lexeme is used in a sentence is determined by its grammatical category. Lexeme can be individual word or multiword.
- For example, the word talk is an example of an individual word lexeme, which may have many grammatical variants like talks, talked and talking. A word like **uneasy** can be broken into two sub-word tokens as **un-easy**.
- A multiword lexeme can be made up of more than one orthographic word. For example, speak up, pull through, etc. are examples of multiword lexemes.



2. Syntax Analysis (Parsing):

- It is the second phase of NLP.
- The purpose of this phase is twofold: to check that a sentence is well-formed or not, and to break it up into a structure that shows the syntactic relationships between the different words.
- For example, a sentence like **The school goes to the boy** would be rejected by a syntax analyzer or parser.
- This phase is used to check grammar, word arrangements, and to show the relationship among the words.

3. Semantic Analysis:

- It is the third phase of NLP.
- Semantic analysis is concerned with the meaning representation. It mainly focuses on the literal meaning of words, phrases, and sentences. The text is checked for meaningfulness. The semantic analyzer disregards sentence such as "hot ice-cream".
- Another Example is "Manhattan calls out to Dave" passes a syntactic analysis because it's a grammatically correct sentence. However, it fails a semantic analysis. Because Manhattan is a place (and can't literally call out to people), the sentence's meaning doesn't make sense.

4. Discourse Integration:

- Discourse Integration depends upon the sentences that precedes it and also invokes the meaning of the sentences that follow it.
- For instance, if one sentence reads, "Manhattan speaks to all its people," and the following sentence reads, "It calls out to Dave," discourse integration checks the first sentence for context to understand that "It" in the latter sentence refers to Manhattan.

5. Pragmatic Analysis:

- It is the fifth phase of NLP. Pragmatic analysis simply fits the actual objects/events, which exist in a given context with object references obtained during the last phase.
- During this, what was said is re-interpreted on what it actually meant. It involves deriving those aspects of language which require real world knowledge.
- For instance, a pragmatic analysis can uncover the intended meaning of "Manhattan speaks to all its people." Methods like neural networks assess the context to understand that the sentence isn't literal, and most people won't interpret it as such. A pragmatic analysis deduces that this sentence is a metaphor for how people emotionally connect with place.
- For example, the sentence Put the banana in the basket on the shelf can have two semantic interpretations and pragmatic analyzer will choose between these two possibilities.

1.5. LANGUAGE MODELING

Language modeling is a fundamental task in natural language processing (NLP) that involves predicting the next word in a sequence of words. The goal is to capture the statistical structure of language and generate coherent and contextually relevant text.

Here's how language modeling typically works:

1. **Input Sequence**: A language model takes as input a sequence of words or tokens. This sequence can be a sentence, paragraph, or longer text.
2. **Context Encoding**: The input sequence is encoded into a numerical representation that can be processed by the language model. This encoding captures the contextual information of the input, such as the meaning of words and their relationships within the sequence.
3. **Prediction**: Based on the encoded context, the language model predicts the probability distribution over the vocabulary of possible next words. This distribution indicates the likelihood of each word occurring given the context provided by the input sequence.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

4. **Sampling**: To generate text, the language model can either select the word with the highest probability (greedy decoding) or sample from the probability distribution to introduce randomness and generate diverse text.

Language modeling can be approached using different techniques:

1. **Statistical Language Models**: These models estimate the probability of word sequences based on statistical analysis of large text corpora. Techniques such as n-grams and smoothing methods are commonly used in statistical language modeling.
2. **Neural Language Models**: Neural network-based approaches, particularly recurrent neural networks (RNNs) and more recently transformer-based architectures like GPT (Generative Pre-trained Transformer), have become prevalent for language modeling. These models learn distributed representations of words and capture long-range dependencies in text.
3. **Fine-tuning**: Pre-trained language models can be fine-tuned on specific tasks or domains to improve performance on downstream tasks like text generation, machine translation, and sentiment analysis. This fine-tuning process adapts the pre-trained model to the characteristics of the target dataset or task.

Language modeling has numerous applications in NLP, including:

- Text generation: Generating coherent and contextually relevant text, such as in chatbots, language generation systems, and content creation tools.
- Machine translation: Modeling the probability of target language words given the source language context.
- Speech recognition: Estimating the likelihood of spoken words or phrases given acoustic features.
- Information retrieval: Ranking documents based on their relevance to a query by modeling the likelihood of word sequences in documents.
- Summarization: Generating concise summaries of longer texts by predicting the most important words or phrases.

Overall, language modeling plays a crucial role in various NLP tasks and continues to be an active area of research and development.

1.5.1. GRAMMAR-BASED LANGUAGE MODELS

Grammar-based language models (LMs) are a class of language models that rely on explicit grammar rules to generate or understand natural language text. These models are based on linguistic theories and formal grammars, which define the syntax and structure of a language.

Here's how grammar-based language models typically work:

1. **Grammar Rules**: Grammar-based LMs start with a set of grammar rules that describe the syntactic structure of the language. These rules define how words and phrases can be combined to form grammatically correct sentences.
2. **Parsing**: When generating or understanding text, the input is parsed according to the grammar rules to identify the syntactic structure of the input sentence. This involves breaking down the input into its constituent parts, such as words, phrases, and clauses, and determining how they relate to each other.
3. **Rule Application**: The grammar rules are then applied to the parsed input to generate or interpret text. These rules govern how words and phrases can be combined to form valid sentences according to the grammar of the language.
4. **Constraints**: Grammar-based LMs may incorporate additional constraints to ensure that the generated text adheres to specific criteria, such as style, domain-specific vocabulary, or semantic coherence.

5. **Evaluation**: The generated text is evaluated based on its grammaticality and coherence according to the rules of the grammar. This evaluation may involve checking for violations of grammar rules, semantic inconsistencies, or other linguistic criteria.

Grammar-based language models have several advantages:

- **Explicit Linguistic Knowledge**: By encoding linguistic knowledge in the form of grammar rules, these models can capture intricate syntactic structures and language patterns.
- **Interpretability**: Since the grammar rules are explicitly defined, the behavior of grammar-based LMs is often more interpretable compared to black-box models like neural networks.

However, grammar-based LMs also have limitations:

- **Scalability**: Creating comprehensive grammar rules for natural languages can be challenging, especially for languages with complex syntax and semantics. As a result, grammar-based LMs may struggle to handle the full richness of natural language.
- **Coverage**: Grammar-based LMs may not capture all linguistic phenomena, leading to gaps in coverage and potential errors in text generation or interpretation.

Overall, while grammar-based language models provide a principled approach to natural language processing, they are often supplemented or replaced by data-driven approaches, such as statistical language models and neural language models, which can learn patterns directly from data without relying on explicit grammar rules.

Grammar-Based Language Modeling Using Context-Free Grammar (CFG) And A Probabilistic Context-Free Grammar (PCFG)

Examples:

S -> NP VP
NP -> Det N
VP -> V NP | V NP PP
PP -> P NP
Det -> "the" | "a"
N -> "cat" | "dog" | "ball"
V -> "chased" | "ate"
P -> "on" | "under" | "with"

This grammar consists of rules for generating

1. sentences (S),
2. noun phrases (NP),
3. verb phrases (VP),
4. prepositional phrases (PP),
5. determiners (Det),
6. nouns (N),
7. verbs (V),
8. prepositions (P).

We start with the start symbol "S" and recursively apply the production rules until we derive a complete sentence:

1. **S**
2. **NP VP** (using the rule S -> NP VP)
3. **Det N VP** (using the rule NP -> Det N)
4. **"the" N VP** (using the rule Det -> "the")
5. **"the" N V NP** (using the rule VP -> V NP)
6. **"the" N V Det N** (using the rule NP -> Det N)

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

7. **"the" N V Det N PP** (using the rule VP -> V NP PP)
8. **"the" N V Det N P NP** (using the rule PP -> P NP)
9. **"the" N V Det N P Det N** (using the rule NP -> Det N)

Now we have a complete sentence: "the cat chased the dog on a ball."

1.5.2. STATISTICAL LANGUAGE MODELLING

Statistical language modeling is a technique used to estimate the probability distribution of word sequences in a language based on observed data. It forms the basis for many natural language processing tasks, such as speech recognition, machine translation, and text generation.

Here's how statistical language modeling typically works:

1. **Training Data**: Statistical language models are trained on large amounts of text data, known as a corpus. This corpus contains sequences of words along with their frequencies of occurrence.
2. **n-gram Models**: One of the simplest approaches to statistical language modeling is the n-gram model, where the probability of a word sequence is estimated based on the frequencies of occurrence of n-length sequences of words (n-grams) in the training data. For example, a bigram model (n=2) estimates the probability of a word given its preceding word, while a trigram model (n=3) estimates the probability of a word given its two preceding words.
3. **Estimating Probabilities**: Given a sequence of words w1, w2, ..., wn, the probability of the entire sequence P(w1, w2, ..., wn) is estimated as the product of the conditional probabilities of each word given its preceding context:
   P(w1, w2, ..., wn) ≈ P(w1) * P(w2|w1) * P(w3|w1, w2) * ... * P(wn|wn−1, ..., w1)
   These probabilities are estimated from the frequencies of n-grams in the training data using techniques such as maximum likelihood estimation (MLE) or smoothed estimation methods like add-one smoothing or Kneser-Ney smoothing.
4. **Backoff and Interpolation**: To address data sparsity issues and improve the robustness of n-gram models, techniques like backoff and interpolation are often employed. Backoff involves using lower-order n-grams when higher-order n-grams have zero counts, while interpolation combines probabilities from different n-gram orders to smooth the probability estimates.
5. **Application**: Once trained, a statistical language model can be used for various NLP tasks. For example, in speech recognition, the language model helps to recognize the most likely sequence of words given the input speech signal. In machine translation, it guides the generation of fluent and grammatically correct translations.

Statistical language modeling provides a simple yet effective framework for capturing the statistical properties of natural language. However, it has limitations such as the inability to capture long-range dependencies and the need for large amounts of training data to achieve good performance. More sophisticated approaches, such as neural language models, have been developed to address these limitations and achieve state-of-the-art results in many NLP tasks.

1. **Speech Recognition**: In speech recognition systems, statistical language models are used to decode the most likely sequence of words given an input speech signal. The language model helps to distinguish between alternative word sequences and improve the accuracy of the recognized text. For example, given the audio signal "I want to eat," the language model may help decide between "I want two eat" and "I want to eat."
2. **Autocomplete and Text Prediction**: Statistical language models power autocomplete and text prediction features in applications such as search engines, messaging apps, and word processors. These models suggest the most likely next word or phrase based on the context of the input text. For example, when typing "I am going to," the language model may suggest "the store" or "the park" as likely completions.

3. **Machine Translation**: In machine translation systems, statistical language models help generate fluent and grammatically correct translations by estimating the probability of different word sequences in the target language. The language model guides the selection of the most likely translation given the source text. For example, given the source text "Je suis content" in French, the language model may help choose between "I am happy" and "I am satisfied" as translations.
4. **Text Generation**: Statistical language models can be used to generate coherent and contextually relevant text for various applications, such as chatbots, content creation tools, and language generation systems. These models estimate the probability of word sequences and use sampling techniques to generate new text based on the learned language patterns. For example, a language model trained on news articles may generate new headlines or article summaries.
5. **Spell Checking and Correction**: Statistical language models are used in spell checking and correction systems to identify and correct spelling errors in text. These models estimate the likelihood of word sequences and suggest corrections based on the context of the input text. For example, when detecting the misspelled word "recieve," the language model may suggest "receive" as a likely correction based on its training data.

These examples demonstrate how statistical language modeling is applied in various NLP tasks to improve the accuracy, fluency, and naturalness of text processing and generation. A simple example of statistical language modeling using a bigram model. Suppose we have a small corpus consisting of the following sentences:

1. "I like to eat apples."
2. "Apples are delicious."
3. "I like to eat bananas."

We can use this corpus to build a bigram language model, which estimates the probability of each word given its preceding word. Here's how we can do it:

1. **Tokenization**: First, we tokenize the sentences into individual words, removing punctuation and converting everything to lowercase. This gives us the following tokenized corpus:
   ["i", "like", "to", "eat", "apples"]
   ["apples", "are", "delicious"]
   ["i", "like", "to", "eat", "bananas"]
2. **Counting Bigrams**: Next, we count the occurrences of bigrams (pairs of consecutive words) in the tokenized corpus:
   ("i", "like"): 2
   ("like", "to"): 2
   ("to", "eat"): 2
   ("eat", "apples"): 1
   ("apples", "are"): 1
   ("are", "delicious"): 1
   ("eat", "bananas"): 1
3. **Estimating Probabilities**: We calculate the probability of each word given its preceding word using maximum likelihood estimation (MLE)
   P("like" | "i") = Count("i like") / Count("i") = 2 / 2 = 1.0
   P("to" | "like") = Count("like to") / Count("like") = 2 / 2 = 1.0
   P("eat" | "to") = Count("to eat") / Count("to") = 2 / 2 = 1.0
   P("apples" | "eat") = Count("eat apples") / Count("eat") = 1 / 2 = 0.5
   P("are" | "apples") = Count("apples are") / Count("apples") = 1 / 1 = 1.0
   P("delicious" | "are") = Count("are delicious") / Count("are") = 1 / 1 = 1.0

P("bananas" | "eat") = Count("eat bananas") / Count("eat") = 1 / 2 = 0.5

Now, we have a bigram language model that can estimate the probability of word sequences. For example, if we want to compute the probability of the sentence "I like to eat bananas," we can multiply the probabilities of the bigrams:

P("i") * P("like" | "i") * P("to" | "like") * P("eat" | "to") * P("bananas" | "eat")
= 1.0 * 1.0 * 1.0 * 1.0 * 0.5
= 0.5

This shows that according to our bigram model, the probability of the sentence "I like to eat bananas" is 0.5.

1.6. REGULAR EXPRESSIONS

Regular Expression (RE) is a formal language used to describe patterns in text. It provides a compact and powerful way to search, match, and manipulate strings. Regular expressions are widely used in text editors, programming languages, UNIX tools (grep, Perl, Emacs), Microsoft Word, and web search engines.

Beyond practical usage, regular expressions play a crucial role in computer science theory and computational linguistics. They form the foundation for pattern recognition systems and are implemented using Finite State Automata (FSA).

Regular expressions were introduced by Stephen Kleene in 1956. Formally, a regular expression is an algebraic notation that defines a set of strings, also called a regular language.

A string is a sequence of symbols such as letters, digits, spaces, punctuation, and special characters. In text processing, even spaces and tabs are treated as characters.

Regular expressions are implemented using Finite State Automata, which are simple machines capable of recognizing patterns. Variants of automata such as Finite-State Transducers, Hidden Markov Models, and N-gram grammars are extensively used in applications like:

● Speech recognition ● Machine translation ● Spell checking ● Information extraction

1.6.1 Basic Regular Expression Patterns

The simplest regular expression is a sequence of ordinary characters.

Example: /woodchuck/

This pattern matches any line containing the exact substring *woodchuck*.

Regular expressions are usually written between slashes / /, following Perl notation. The slashes help distinguish the pattern from normal text but are not part of the expression itself.

A regular expression may consist of:

- A single character (e.g., /!/)
- A sequence of characters (e.g., /urgl/)

Most search tools return the line containing the match, often highlighting the first occurrence.

Case Sensitivity

Regular expressions are case-sensitive by default.

Example:

- /s/ matches s
- /S/ matches S

Thus, /woodchucks/ does not match Woodchucks. To handle capitalization, we use character classes.

RE	Example Patterns Matched
/woodchucks/	"interesting links to woodchucks and lemurs"
/a/	"Mary Ann stopped by Mona's"
/Claire␣says,/	" "Dagmar, my gift please," Claire says,"
/DOROTHY/	"SURRENDER DOROTHY"
/!/	"You've left the burglar behind again!" said Nori

Character Classes and Ranges

Square Brackets [ ]

Square brackets represent a set of alternative characters, meaning OR.

Example: /[wW]/

Matches either w or W.

/[0-9]/

Matches any single digit.

RE	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	"Woodchuck"
/[abc]/	'a', 'b', *or* 'c'	"In uomini, in soldati"
/[1234567890]/	any digit	"plenty of 7 to 5"

Figure 2.1 The use of the brackets [] to specify a disjunction of characters.

Ranges Using -

Ranges allow us to specify a sequence of characters concisely.

Examples:

- /[a-z]/ → any lowercase letter
- /[A-Z]/ → any uppercase letter
- /[2-5]/ → digits 2 through 5

RE	Match	Example Patterns Matched
/[A-Z]/	an uppercase letter	"we should call it 'Drenched Blossoms'"
/[a-z]/	a lowercase letter	"my beans were impatient to be hoed!"
/[0-9]/	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.2 The use of the brackets [] plus the dash - to specify a range.

Negated Character Classes [^]

If the caret ^ appears immediately after [, the class is negated.

Example: /[^a-zA-Z]/

Matches any character except letters.

Note: The caret has different meanings depending on context.

Optional and Repetition Operators

Question Mark ?

The ? operator means zero or one occurrence of the preceding character.

Example: /woodchucks?/

Matches:

- woodchuck
- woodchucks

Kleene Star *

The Kleene star means zero or more occurrences of the previous character or group.

Example: /a*/

Matches:

- Empty string
- a

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

- aaaa

To match one or more occurrences: /aa*/

Kleene Plus +

The Kleene plus operator means one or more occurrences.

Example: /[0-9]+/ matches a sequence of digits.

Wildcards and Anchors

Wildcard .

The dot (.) matches any single character except newline.

Example: /a.b/ matches aab, a3b, a_b.

Combined with *:

/.*/ matches any string.

RE	Match	Example Patterns
/beg.n/	any character between *beg* and *n*	begin, beg'n, begun

Figure 2.5 The use of the period . to specify any character.

Anchors

Anchors match positions, not characters.

- ^ → start of a line
- $ → end of a line

Example: /^The/

Matches *The* only at the beginning of a line.

/^The dog\.$/ matches exactly the line *The dog.*

Word Boundaries

- \b → word boundary
- \B → non-boundary

Example: /\bthe\b/ matches *the* but not *other*.

RE	Match (single characters)	Example Patterns Matched
[^A-Z]	not an uppercase letter	"Oyfn pripetchik"
[^Ss]	neither 'S' nor 's'	"I have no exquisite reason for't"
[^\.]	not a period	"our resident Djinn"
[e^]	either 'e' or '^'	"look up ^ now"
a^b	the pattern 'a^b'	"look up a^b now"

Figure 2.3 Uses of the caret ^ for negation or just to mean ^ . We'll discuss below the need to escape the period by a backslash.

RE	Match	Example Patterns Matched
woodchucks?	woodchuck or woodchucks	"woodchuck"
colou?r	color or colour	"colour"

Figure 2.4 The question-mark ? marks optionality of the previous expression.

1.6.2 Disjunction, Grouping, and Precedence

Parenthesis	()
Counters	* + ? {}
Sequences and anchors	the ^my end$
Disjunction	|

Disjunction |

The pipe symbol represents OR.

Example: /cat|dog/

Square brackets cannot be used here because they match only single characters, not strings.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

Grouping with Parentheses ( )

Parentheses group patterns so operators apply to the whole group.

Example: /gupp(y|ies)/ matches *guppy* and *guppies*.

Operator Precedence

From highest to lowest:

1. Counters (*, +, ?, {})
2. Sequence (concatenation)
3. Disjunction (|)

Regular expressions are greedy, meaning they match the longest possible string.

Error Analysis in Regex Design

When designing regular expressions, two types of errors may occur:

- False Positives: Incorrect matches (e.g., matching *the* in *other*)
- False Negatives: Missing valid matches (e.g., missing *The*)

Regex design involves balancing:

- Accuracy (reducing false positives)
- Coverage (reducing false negatives)

1.6.3 Advanced Regular Expression Operators

Character Class Aliases

- Regular expressions provide aliases (short forms) for common character ranges.
- These are used to reduce typing and improve readability.
  - Examples (conceptual): Digits, letters, whitespace, etc.
- They behave the same as their expanded ranges (e.g., digits = [0–9]).

Explicit Counters { }

- {n} → exactly n times
- {n,m} → n to m times
- {n,} → at least n times

Example: /[0-9]{3}/ matches exactly three digits.

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	*n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression

Figure 2.7 Regular expression operators for counting.

Escape Sequences

Some characters have special meanings and must be escaped.

Examples:

- \n → newline
- \t → tab
- \. → literal dot
- * → literal star
- \\ → backslash

RE	Expansion	Match	Examples
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r␣\t␣\n␣\f␣]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in␣Concord

Figure 2.6 Aliases for common sets of characters.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

1.7 FINITE-STATE AUTOMATA

- Regular expressions are not just for text searching; they are closely related to finite-state automata (FSA).
- Any regular expression can be converted into an FSA, and any FSA can be represented as a regular expression.
- Regular expressions define a class of formal languages called **regular languages**, which can also be described by FSAs.
- Regular grammars provide a third equivalent way to represent regular languages.

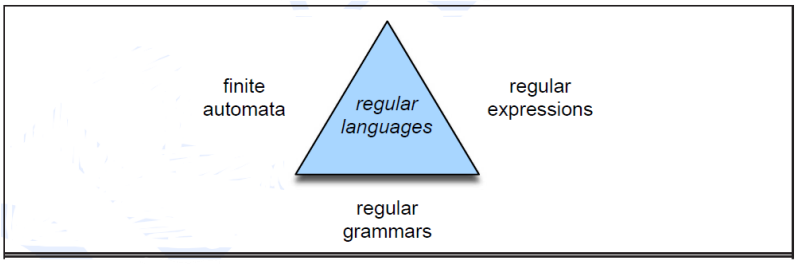


Figure 1.1 - Finite automata, regular expressions, and regular grammars are all equivalent ways of describing regular languages

- A finite-state automaton (FSA) is used to recognize strings.
- The input string is read symbol by symbol from a tape.
- An FSA is represented as a directed graph with:
  - States (circles) and Transitions (arrows)
- The machine starts at the initial state (q_0) and:
  - Reads the next symbol
  - Follows the matching transition
  - Moves to the next state
- If the machine reaches the accepting state when the input ends, the string is accepted.
- The string is rejected if:
  - Input ends too early
  - No valid transition exists
  - The machine stops in a non-accepting state

State-Transition Table

- An automaton can also be represented using a state-transition table.
  This table shows:
  - The start state, the accepting (final) states, and the next state for each input symbol.

1.7.1 Using an FSA to Recognize Sheeptalk

We defined the sheep language as any string from the following (infinite) set.

baa!
baaa!
baaaa!
baaaaa!
……..

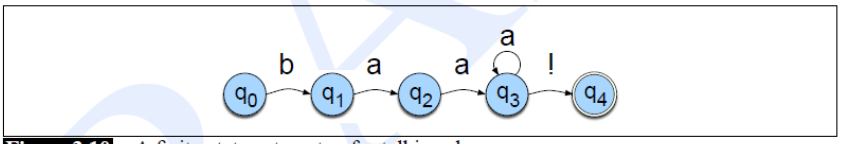


Figure 2.10 A finite-state automaton for talking sheep.

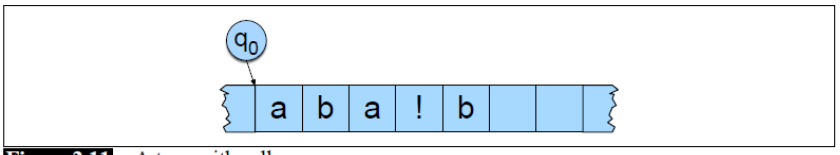


Figure 2.11 A tape with cells.

- The regular expression for sheep talk is **/baa+!/**, which can be modeled using a finite automaton.
- A finite-state automaton (FSA) recognizes the same set of strings as the regular expression.
- The automaton has five states:
  - State 0 is the start state and State 4 is the final (accepting) state, shown with a double circle
- The automaton includes four transitions between states.
- In the transition table:
  - A colon (:) marks the final state and Ø shows an invalid transition
- For example, from state 0:
  - Reading **b** moves to state 1
  - Reading **a** or **!** causes rejection

	Input		
State	b	a	!
0	1	Ø	Ø
1	Ø	2	Ø
2	Ø	3	Ø
3	Ø	3	4
4:	Ø	Ø	Ø

Formal Definition of a Finite Automaton

A finite automaton is formally defined by five components:

$Q = q_0 q_1 q_2 \ldots q_{N-1}$	a finite set of N **states**
Σ	a finite **input alphabet** of symbols
q_0	the **start state**
F	the set of **final states**, $F \subseteq Q$
$\delta(q,i)$	the **transition function** or transition matrix between states. Given a state $q \in Q$ and an input symbol $i \in \Sigma$, $\delta(q,i)$ returns a new state $q' \in Q$. δ is thus a relation from $Q \times \Sigma$ to Q;

For the sheeptalk automaton shown in Fig. 2.10:

- Q = {q_0, q_1, q_2, q_3, q_4} → set of states
- Σ = {a, b, !} → input alphabet
- F = {q_4} → set of accepting states
- δ(q, i) → transition function, defined by the state-transition table

Deterministic Recognition Algorithm (D-RECOGNIZE)

- Figure 2.12 shows an algorithm called D-RECOGNIZE (Deterministic Recognizer). It is called deterministic because there is exactly one possible action for each input symbol in every state - the algorithm never has to guess.
- D-RECOGNIZE takes:
  - an input tape, and
  - a finite automaton

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

- It returns accept if the string on the tape is recognized by the automaton; otherwise, it returns reject.
- Fig. 2.13 traces the execution of this algorithm on the sheep language FSA given the sample input string baaa !.

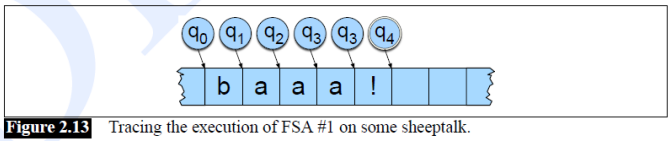


Figure 2.13 Tracing the execution of FSA #1 on some sheeptalk.

- Before reading the input tape, the machine starts in state q_0.
- When it reads b, it moves to state q_1, as specified by the transition table entry $\delta(q_0, b)$.
- Next, it reads a and moves to state q_2.
- Reading another a moves it to state q_3.
- A third a keeps the machine in state q_3.
- When it reads !, it moves to state q_4.
- At this point, there is no more input, so the machine checks the end-of-input condition and halts in state q_4. Since q_4 is an accepting state, the string baaa! is accepted as a valid sentence in the sheep language.

```
function D-RECOGNIZE(tape, machine) returns accept or reject

  index ← Beginning of tape
  current-state ← Initial state of machine
  loop
   if End of input has been reached then
     if current-state is an accept state then
       return accept
     else
       return reject
   elsif transition-table[current-state,tape[index]] is empty then
     return reject
   else
     current-state ← transition-table[current-state,tape[index]]
     index ← index + 1
  end
```

Figure 2.12 An algorithm for deterministic recognition of FSAs. This algorithm returns *accept* if the entire string it is pointing at is in the language defined by the FSA, and *reject* if the string is not in the language.

Failure and Rejecting Inputs

- The algorithm fails if there's no valid transition for a state and input symbol.
- Example: abc is rejected because q_0 has no transition on a, and c is outside the sheep-talk alphabet.

Fail (Sink) State

- Missing or illegal transitions lead to a special **fail (sink) state, qF**.
- Once entered, the machine stays in qF and rejects all further input.
- Any automaton with missing transitions can be converted to an equivalent one with:
  - A fail state, and
  - Transitions from all undefined state–input pairs to qF.
- For completeness, Fig. 2.14 shows the sheep-talk automaton with the fail state qF added.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

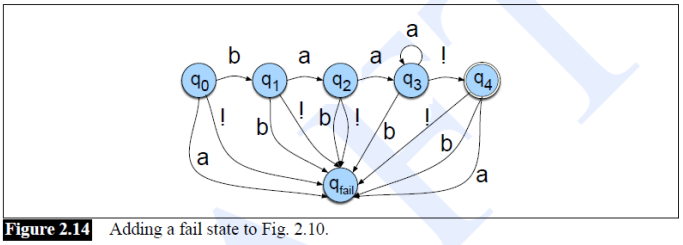


Figure 2.14 Adding a fail state to Fig. 2.10.

1.7.2 Formal Languages

A formal language is a set of strings, where each string is made using symbols from a finite alphabet (Σ). The same alphabet used in finite-state automata (FSA) is also used to define formal languages. A model that can both generate and recognize all valid strings acts as the definition of a formal language.

Alphabet example, in the sheep language, the alphabet is: Σ = { a, b, ! }

Language Defined by a Model

A model such as an FSA defines a language. The notation L(m) means the language described by model *m*. For example, if *m* is the sheeptalk automaton, then:

L(m) = { baa!, baaa!, baaaa!, baaaaa!, ... }. This language is infinite.

Use of Automata

Automata provide a compact way to describe infinite languages. Instead of listing all possible strings, they generate or recognize them systematically.

Formal vs Natural Languages

Formal languages are different from natural languages like English or Tamil. They may not look like human language and are often used to model systems such as machines or processes.

Role in Linguistics

- Formal languages help model parts of natural language, including:
  - Phonology • Morphology • Syntax
- A generative grammar is a set of rules or automata that defines a formal language and produces all valid strings in that language.

1.7.3 Non-Deterministic Finite State Automata (NFSAs)

In deterministic finite state automata (DFSAs), the next state is uniquely determined by the current state and the input symbol. Consider the sheeptalk automaton shown in Fig. 2.17.

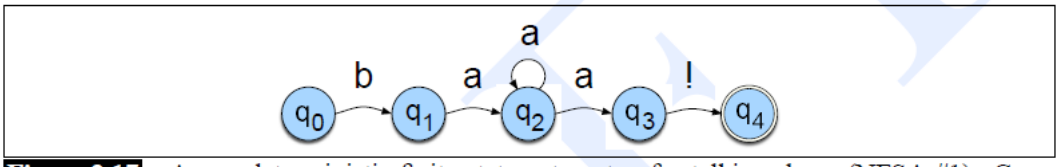


Figure 2.17 A non-deterministic finite-state automaton for talking sheep (NFSA #1). Compare with the deterministic automaton in Fig. 2.10.

It is similar to the earlier automaton in Fig. 2.10, except that the self-loop is on state 2 instead of state 3. When the machine reaches state 2 and reads the input symbol a, there are two possible transitions:

- Stay in state 2, or
- Move to state 3

Because the machine cannot uniquely decide which transition to take, this automaton is called a non-deterministic FSA (NFSA).

In contrast, the automaton in Fig. 2.10 is deterministic, since for every state and input symbol there is exactly one possible transition. Such automata are called DFSAs.

Non-Determinism Due to ϵ-Transitions

Another common source of non-determinism is the presence of ϵ-transitions (epsilon transitions). In Fig. 2.18, the automaton recognizes the same language as the earlier sheeptalk automata, but it includes an ϵ-transition.

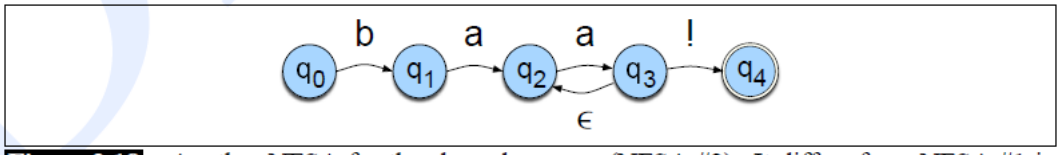


Figure 2.18 Another NFSA for the sheep language (NFSA #2). It differs from NFSA #1 in Fig. 2.17 in having an ϵ-transition.

An ϵ-transition allows the automaton to:

- Move from one state to another without consuming any input symbol, and
- Without advancing the input pointer

For example, if the machine is in state 3, it can move to state 2 either by:

- Taking the ϵ-transition, or
- Following the transition labeled !

Since the machine has multiple possible moves without reading input, this also introduces non-determinism.

1.7.4 Using a Non-Deterministic FSA (NFSA) to Accept Strings

A Non-Deterministic Finite State Automaton (NFSA) is used to recognize strings where more than one transition may be possible for the same input symbol. At certain states, the machine can choose between multiple paths. If it follows a wrong path, it may temporarily reject a string that should actually be accepted. This problem of multiple choices is known as **non-determinism** and commonly occurs in parsing and pattern recognition.

Ways to Handle Non-Determinism

There are three common methods:

1. Backup (Backtracking)
   - The machine remembers the current state and input position.
   - If one path fails, it goes back and tries another path.
2. Look-ahead
   - The machine checks upcoming input symbols to choose the correct transition.
3. Parallelism
   - All possible transitions are explored at the same time.

Changes in Transition Table for NFSA

To support non-determinism:

• An extra column is added for **ε-transitions** (moves without consuming input).

• Each cell may contain **multiple next states** instead of one.

Example: From state q_2 with input 'a', the automaton may go to q_2 or q_3.

The transition table shown for NFSA #1 (Fig. 2.17) illustrates this behaviour.

	Input			
State	b	a	!	ϵ
0	1	0	0	0
1	0	2	0	0
2	0	2,3	0	0
3	0	0	4	0
4:	0	0	0	0

ND-RECOGNIZE Algorithm

This algorithm is used to recognize strings using an NFSA.

Main Concepts:

• Uses an agenda to store unexplored choices

• Each choice is a search-state = (current state, input position)

Delayed Rejection

Unlike deterministic automata:

• NFSA does NOT reject immediately when a path fails
• It only rejects after all possible paths are tried

Example: Input "baaa!"

At state q_2 with input 'a', two transitions exist:

✓ Stay in q_2 (loop)

✓ Move to q_3

Both choices are stored and explored.

Even if one fails, the other may succeed and reach the accepting state q_4.

```
function ND-RECOGNIZE(tape, machine) returns accept or reject

  agenda ← {(Initial state of machine, beginning of tape)}
  current-search-state ← NEXT(agenda)
  loop
    if ACCEPT-STATE?(current-search-state) returns true then
      return accept
    else
      agenda ← agenda ∪ GENERATE-NEW-STATES(current-search-state)
    if agenda is empty then
      return reject
    else
      current-search-state ← NEXT(agenda)
  end
function GENERATE-NEW-STATES(current-state) returns a set of search-states

  current-node ← the node the current search-state is in
  index ← the point on the tape the current search-state is looking at
  return a list of search states from transition table as follows:
    (transition-table[current-node,ϵ], index)
    ∪
    (transition-table[current-node, tape[index]], index + 1)
function ACCEPT-STATE?(search-state) returns true or false

  current-node ← the node search-state is in
  index ← the point on the tape search-state is looking at
  if index is at the end of the tape and current-node is an accept state of machine
  then
    return true
  else
    return false
```

Figure 2.19 An algorithm for NFSA recognition. The word *node* means a state of the FSA, while *state* or *search-state* means "the state of the search process", i.e., a combination of *node* and *tape-position*.

1.7.5 Regular Languages and Finite-State Automata

Regular languages are a class of languages that can be described using regular expressions and can also be recognized by finite-state automata (FSA), which may be deterministic (DFA) or non-deterministic (NFA). Since both regular expressions and FSAs define the same set of languages, these languages are known as regular languages.

Alphabet and Basic Concepts

- An alphabet (Σ) is a finite set of symbols used to form strings.
- The empty string (ε) represents a string of length zero and is not a symbol in the alphabet, while the empty set (∅) represents a language with no strings and is different from the empty string.
- Regular languages are built using symbols from Σ along with ε and ∅.

Construction of Regular Languages

- Regular languages are formed using operations such as concatenation, union (disjunction), and Kleene star (repetition).
- All regular expression operators can be reduced to these three basic operations.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

- For example, square brackets represent union (such as [ab] meaning a or b), and counters like + or {n,m} are special cases of repetition.

Closure Properties

Regular languages are closed under operations like union, concatenation, and repetition, which means applying these operations to regular languages always results in another regular language.

Equivalence of Regular Expressions and FSAs

Regular expressions and finite-state automata are equivalent in power. For every regular expression, an equivalent automaton can be built, and for every automaton, an equivalent regular expression can be constructed.

Construction of Automata from Regular Expressions

The construction begins with simple base cases such as the empty set (∅), empty string (ε), and single symbols from the alphabet. More complex automata are formed using operations like concatenation, union, and Kleene star by combining smaller automata.

Concatenation Operation in FSA

In concatenation, the final states of the first automaton are connected to the initial state of the second automaton using ε-transitions.

Kleene Star Operation in FSA

To implement repetition, a new initial state and a new final state are created. ε-transitions are added from the new initial state to the old initial state, from old final states back to the old initial state to allow repetition, and directly from the new initial state to the new final state to allow zero occurrences.

Union Operation in FSA

For union, a new initial state is added and ε-transitions are created from it to the initial states of the automata being combined.

1.8 ENGLISH MORPHOLOGY

- Morphology: Study of how words are built from morphemes (smallest meaning-bearing units).
  - Examples: *fox* = 1 morpheme; *cats* = 2 morphemes (*cat* + *-s*).
  - Stem vs. Affix:
    - Stem: main meaning of the word.
    - Affix: adds additional meaning.
- Types of affixes:
  Prefix (before stem), suffix (after stem), infix (inside stem), circumfix (around stem).
- Examples:
  - *eats* = stem *eat* + suffix *-s*
  - *unbuckle* = prefix *un-* + stem *buckle*

Prefixes and Suffixes in Other Languages

- In some languages, inflection is shown using both prefixes and suffixes.
- For example, in **German**, the past participle of certain verbs is formed by adding the prefix **ge-** at the beginning of the verb stem and the suffix **-t** at the end.
- Thus, the verb *sagen* (to say) becomes *gesagt* (said).

Infixes in Languages

- An **infix** is a morpheme that is inserted into the middle of a word.
- Infixes are common in some languages, such as **Tagalog**, a Philippine language.
- For example, the infix **-um-**, which marks the agent of an action, is inserted into the stem *hingi* (borrow) to form *humingi*.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

Infixes in English

- English rarely uses infixes, but a limited form exists in informal speech.
- In some dialects, taboo or emphatic morphemes such as **"f**king"** or **"bl**dy"** are inserted into the middle of words for emphasis.
- Examples include *Man-f**king-hattan* and *abso-bl**dy-lutely*.
- These forms are non-standard and mainly used in spoken language.

- A word can contain more than one **affix**.
  - Example: *rewrites* = **re-** (prefix) + **write** (stem) + **-s** (suffix).
  - Example: *unbelievably* = **believe** (stem) + **un-**, **-able**, **-ly** (affixes).
- English usually has only a few affixes in a word, but some languages like **Turkish** can have many affixes (nine or ten) attached to a single stem.
- Languages that form words by stringing many affixes together are called **agglutinative languages**.
- Words are formed by combining morphemes in different ways. Four important processes used in speech and language processing are:
  Inflection, Derivation, Compounding, and Cliticization.
- **Inflection** adds grammatical information without changing the word class.
  - Example: *cats* (plural **-s**), *walked* (past tense **-ed**).
- **Derivation** forms a new word, often changing the word class and meaning.
  - Example: *computerize* → *computerization* using **-ation**.
- **Compounding** combines two or more stems to form a new word.
  - Example: *dog* + *house* → *doghouse*.
- **Cliticization** attaches a **clitic**, which behaves like a word but is phonologically reduced and attached to another word.
  - Example: *I've* ('ve), French *l'opera* (l').

INFLECTIONAL MORPHOLOGY

- Inflectional morphology studies how words change their form to express grammatical meanings such as number, tense, and possession without changing the basic meaning or word class. English has a relatively simple inflectional system, and only nouns, verbs, and sometimes adjectives take inflectional endings.

Inflection in English Nouns

- English nouns have only two types of inflection: plural and possessive. The plural is usually formed by adding **-s** or **-es** to the noun. The ending **-es** is used after nouns ending in **-s, -z, -sh, -ch,** and sometimes **-x**. Nouns ending in **-y** preceded by a consonant change **-y** to **-ies**. Some nouns form plurals irregularly, such as *child* → *children*.

	Regular Nouns		Irregular Nouns	
Singular	cat	thrush	mouse	ox
Plural	cats	thrushes	mice	oxen

Possessive Form of Nouns

- The possessive form shows ownership. Singular nouns take **'s** (llama's wool). Plural nouns ending in **-s** take only an apostrophe (llamas' wool). Irregular plural nouns take **'s** (children's toys). Some names ending in **-s or -z** may also take only an apostrophe.

Types of Verbs in English

- English verbs are classified into three types: main verbs, modal verbs, and primary verbs. Main verbs include words like *eat* and *sleep*, modal verbs include *can* and *will*, and primary verbs include *be*, *have*, and *do*. Only main and primary verbs take inflectional endings.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

Regular Verbs

- Most English verbs are regular. Regular verbs have four main forms: the base form, the **-s** form, the **-ing** form, and the **-ed** form. These forms are predictable and are created by adding standard endings to the base verb. Regular verbs are productive, meaning new verbs such as *fax* follow the same pattern (*faxed, faxing, faxes*).

Morphological Class	Regularly Inflected Verbs			
stem	walk	merge	try	map
-s form	walks	merges	tries	maps
-ing participle	walking	merging	trying	mapping
Past form or *-ed* participle	walked	merged	tried	mapped

Irregular Verbs

- Irregular verbs do not follow regular inflectional patterns. They may have three to eight different forms. For example, *eat* → *ate* → *eaten* and *cut* → *cut* → *cut*. Although irregular verbs are few in number, they include many of the most frequently used verbs in English.

Morphological Class	Irregularly Inflected Verbs		
stem	eat	catch	cut
-s form	eats	catches	cuts
-ing participle	eating	catching	cutting
preterite	ate	caught	cut
past participle	eaten	caught	cut

Use of Verb Forms

- The **-s** form is used in the present tense for third-person singular subjects. The base form is used in infinitives and after certain verbs. The **-ing** form is used to show ongoing actions and as a gerund, where the verb acts as a noun. The **-ed/-en** form is used in perfect tenses and passive constructions.

Spelling Changes in Verb Inflection

- Certain spelling changes occur when inflectional endings are added. A final consonant may be doubled before adding **-ing** or **-ed**. A silent **-e** is dropped before adding these endings. Verbs ending in **-y** preceded by a consonant change **-y** to **-i**, and verbs ending in **-s, -z, -sh, -ch,** and **-x** take **-es**.

Comparison with Other Languages

- English verb inflection is much simpler than in many other languages. For example, Spanish verbs can have more than fifty different forms for a single regular verb, while English verbs usually have only four or five forms.

	Present Indicative	Imperfect Indicative	Future	Preterite	Present Subjunctive	Conditional	Imperfect Subjunctive	Future Subjunctive
1SG	amo	amaba	amaré	amé	ame	amaría	amara	amare
2SG	amas	amabas	amarás	amaste	ames	amarías	amaras	amares
3SG	ama	amaba	amará	amó	ame	amaría	amara	amáreme
1PL	amamos	amábamos	amaremos	amamos	amemos	amaríamos	amáramos	amáremos
2PL	amáis	amabais	amaréis	amasteis	améis	amaríais	amarais	amareis
3PL	aman	amaban	amarán	amaron	amen	amarían	amaran	amaren

Figure 3.1 To love in Spanish. Some of the inflected forms of the verb *amar* in European Spanish. *1SG* stands for "first person singular", 3PL for "third person plural", and so on.

DERIVATIONAL MORPHOLOGY

- Derivational morphology deals with the formation of **new words** by adding derivational morphemes to a word stem.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

- It often results in a **change of word class** and creates meanings that are not always predictable.
- Compared to inflection, derivation in English is **more complex**.

Nominalization

- **Nominalization** is the process of forming **nouns** from verbs or adjectives.
- It is one of the most common derivational processes in English.
- The suffix **-ation** is frequently used to form nouns, especially from verbs ending in **-ize**.
- Example:
  - computerize → computerization
- Some nominalizing suffixes are highly **productive** but not universal.

Suffix	Base Verb/Adjective	Derived Noun
-ation	computerize (V)	computerization
-ee	appoint (V)	appointee
-er	kill (V)	killer
-ness	fuzzy (A)	fuzziness

Formation of Adjectives

- Adjectives can be derived from **nouns and verbs** using derivational suffixes.
- These suffixes change the word class and add descriptive meaning.
- Such derived adjectives are widely used in English.

Suffix	Base Noun/Verb	Derived Adjective
-al	computation (N)	computational
-able	embrace (V)	embraceable
-less	clue (N)	clueless

Reasons for Complexity in Derivation

- Derivational affixes are **less productive** than inflectional affixes.
- Not all suffixes can be added to every word stem.
- Incorrect forms are marked with an asterisk (*).
- Examples of non-acceptable forms:
  - *eatation
  - *spellation

Meaning Differences in Derivation

- Different derivational suffixes can create **subtle differences in meaning**.
- Words derived from the same stem may not mean exactly the same thing.
  - Example: sincerity ≠ sincereness

CLITICIZATION

- A clitic is a linguistic unit that lies between a word and an affix.
- Clitics are phonologically similar to affixes because they are usually short and unaccented.
- Syntactically, clitics behave more like independent words, often functioning as pronouns, articles, conjunctions, or verbs.

Full Form	Clitic	Full Form	Clitic
am	'm	have	've
are	're	has	's
is	's	had	'd
will	'll	would	'd

Types of Clitics

- Proclitics: clitics that occur before a word.
- Enclitics: clitics that occur after a word.

Clitics in English

- English commonly uses enclitic forms, especially with auxiliary verbs.
- Examples include contractions such as:
  - she's = she is / she has
- These forms are sometimes ambiguous in meaning.
- In English, identifying clitics is usually easy due to the use of the apostrophe.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

Clitics in Other Languages

- Clitics are more difficult to identify in some languages.
- In Arabic and Hebrew, the definite article is a proclitic:
  - Arabic: al-
  - Hebrew: ha-
- These clitics attach to the front of nouns and must be separated for tasks like parsing and part-of-speech tagging.

Proclitics and Enclitics in Arabic

- Arabic has several proclitics, including:
  - Prepositions (e.g., b meaning "by/with")
  - Conjunctions (e.g., w meaning "and")
- Arabic also uses enclitics to mark certain pronouns.
- A single Arabic word can contain:
  - A conjunction, A preposition, A pronoun clitic, A stem, A plural suffix

Direction of Writing

- Arabic is written from right to left.
- Therefore, the order of clitics in written Arabic appears reversed compared to English.

	proclitic	proclitic	stem	affix	enclitic
Arabic	w	b	Hsn	At	hm
Gloss	and	by	virtue	s	their

NON-CONCATENATIVE MORPHOLOGY

- The type of morphology where words are formed by joining morphemes in a linear sequence is called concatenative morphology.
- Most of the morphology discussed earlier (prefixes, suffixes, inflection) belongs to this type.
- Non-concatenative morphology refers to word formation where morphemes are combined in non-linear or complex ways.
- Morphemes may be interwoven or inserted rather than simply attached at the beginning or end of a word.

Infixation as Non-Concatenative Morphology

- Infixation is one type of non-concatenative morphology.
- Example: In Tagalog, the infix -um- is inserted into the stem *hingi* to form *humingi*.
- Here, the morphemes are intermingled, not concatenated.

Templatic (Root-and-Pattern) Morphology

- Another important type of non-concatenative morphology is templatic morphology, also called root-and-pattern morphology.
- This system is common in Semitic languages such as Arabic and Hebrew.

Root and Template Structure

- Words are formed using:
  - A root consisting usually of three consonants (CCC) that carry the basic meaning.
  - A template that specifies vowel patterns and consonant order, adding grammatical and semantic information such as voice or intensity.

Examples from Hebrew

- The Hebrew root lmd means "learn/study".
- Different templates produce different meanings:
  - CaCaC (active) → *lamad* ("he studied")
  - CiCeC (intensive) → *limed* ("he taught")

- CuCaC (intensive passive) → *lumad* ("he was taught")

Combination with Concatenative Morphology

- Languages like Arabic and Hebrew use both:
  - Non-concatenative (templatic) morphology
  - Concatenative morphology (prefixes and suffixes)

1.9 TRANSDUCERS FOR LEXICON AND RULES

Two-Level Morphology System Architecture

- We are now ready to combine our lexicon and rule transducers for parsing and generating.
- Fig. 3.19 shows the architecture of a two-level morphology system, whether used for parsing or generating.
- The lexicon transducer maps between the lexical level, with its stems and morphological features, and an intermediate level that represents a simple concatenation of morphemes.
- Then a host of transducers, each representing a single spelling rule constraint, all run in parallel so as to map between this intermediate level and the surface level.
- Putting all the spelling rules in parallel is a design choice; we could also have chosen to run all the spelling rules in series (as a long cascade), if we slightly changed each rule.

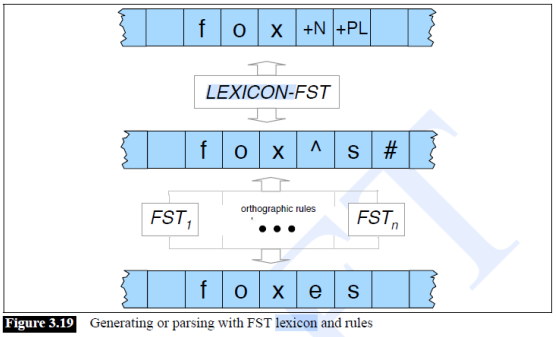


Figure 3.19 Generating or parsing with FST lexicon and rules

Cascade of Transducers

- The architecture in Fig. 3.19 is a two-level cascade of transducers.
- Cascading two automata means running them in series with the output of the first feeding the input to the second.
- Cascades can be of arbitrary depth, and each level might be built out of many individual transducers.
- The cascade in Fig. 3.19 has two transducers in series: the transducer mapping from the lexical to the intermediate levels, and the collection of parallel transducers mapping from the intermediate to the surface level.
- The cascade can be run top-down to generate a string, or bottom-up to parse it; Fig. 3.20 shows a trace of the system accepting the mapping from fox +N +PL to foxes.

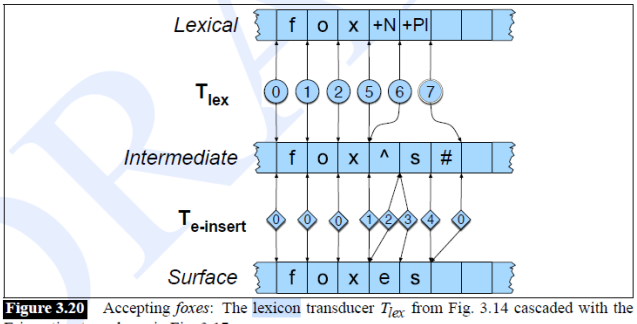


Figure 3.20 Accepting *foxes*: The lexicon transducer T_{lex} from Fig. 3.14 cascaded with the E-insertion transducer in Fig. 3.17.

Power of Finite-State Transducers

- The power of finite-state transducers is that the exact same cascade with the same state sequences is used when the machine is generating the surface tape from the lexical tape, or when it is parsing the lexical tape from the surface tape.
- For example, for generation, imagine leaving the Intermediate and Surface tapes blank.
- Now if we run the lexicon transducer, given fox +N +PL, it will produce foxˆs# on the Intermediate tape via the same states that it accepted the Lexical and Intermediate tapes in our earlier example.
- If we then allow all possible orthographic transducers to run in parallel, we will produce the same surface tape.

Ambiguity in Parsing

- Parsing can be slightly more complicated than generation, because of the problem of ambiguity.
- For example, foxes can also be a verb (albeit a rare one, meaning "to baffle or confuse"), and hence the lexical parse for foxes could be fox +V +3Sg as well as fox +N +PL.
- How are we to know which one is the proper parse?
- In fact, for ambiguous cases of this sort, the transducer is not capable of deciding.
- Disambiguating will require some external evidence such as the surrounding words.
- Thus foxes is likely to be a noun in the sequence *"I saw two foxes yesterday"*, but a verb in the sequence *"That trickster foxes me every time!"*.
- Barring such external evidence, the best our transducer can do is just enumerate the possible choices; so we can transduce foxˆs# into both fox +V +3SG and fox +N +PL.

Local Ambiguity and Search

- There is a kind of ambiguity that we need to handle: local ambiguity that occurs during the process of parsing.
- For example, imagine parsing the input verb *assess*.
- After seeing *ass*, our E-insertion transducer may propose that the e that follows is inserted by the spelling rule.
- It is not until we don't see the # after *asses*, but rather run into another *s*, that we realize we have gone down an incorrect path.
- Because of this non-determinism, FST-parsing algorithms need to incorporate some sort of search algorithm.

Eliminating Spurious Segmentations

Note that many possible spurious segmentations of the input, such as parsing *assess* as *ˆaˆsˆsesˆs*, will be ruled out since no entry in the lexicon will match this string.

Simplifying Cascades

- Running a cascade, particularly one with many levels, can be unwieldy.
- Luckily, we've already seen how to compose a cascade of transducers in series into a single more complex transducer.
- Transducers in parallel can be combined by automaton intersection.
- The automaton intersection algorithm just takes the Cartesian product of the states, i.e., for each state *qi* in machine 1 and state *qj* in machine 2, we create a new state *qi j*.
- Then for any input symbol *a*, if machine 1 would transition to state *qn* and machine 2 would transition to state *qm*, we transition to state *qnm*.
- Fig. 3.21 sketches how this intersection (∧) and composition (◦) process might be carried out.

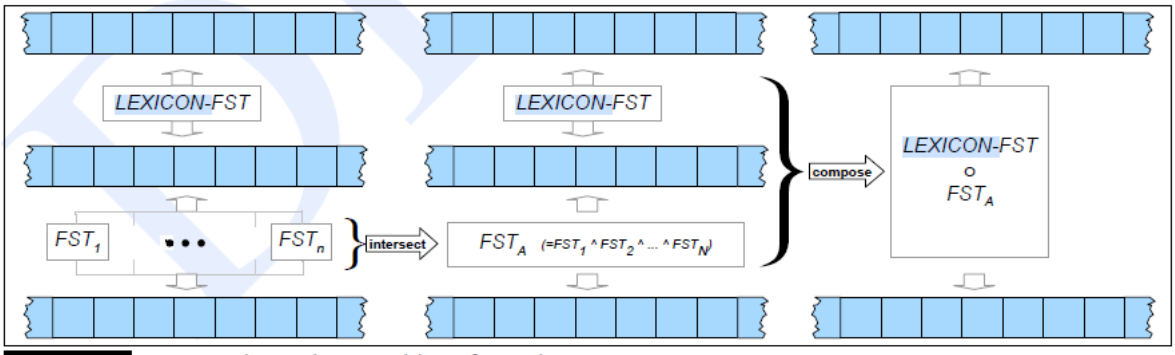


Figure 3.21 Intersection and composition of transducers.

Practical Implementation

Since there are a number of rule→FST compilers, it is almost never necessary in practice to write an FST by hand.

Kaplan and Kay (1994) give the mathematics that define the mapping from rules to two-level relations, and Antworth (1990) gives details of the algorithms for rule compilation.

Mohri (1997) gives algorithms for transducer minimization and determinization.

1.10 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

1.10.1 WORD TOKENIZATION

- Tokenization is the process of dividing running text into meaningful units called tokens and is a basic step in NLP.
- Punctuation is usually retained because it provides useful information such as sentence boundaries and syntactic structure, though word-internal punctuation is preserved (e.g., *Ph.D., AT&T*).
- Numbers, dates, prices, URLs, hashtags, and email addresses must be treated as single tokens (e.g., *$45.55, 01/02/06*).
- Tokenization rules vary across languages, especially for number formats (e.g., *555,500.50* in English vs *555 500,50* in some European languages).
- Tokenizers may also expand clitic contractions (we're → we are) and handle multiword expressions like *New York*, making tokenization closely related to named entity recognition.

Penn Treebank Tokenization

- The Penn Treebank tokenization standard is a widely used method for tokenizing text in NLP.
- It is used in parsed corpora (treebanks) released by the Linguistic Data Consortium (LDC).

Input: "The San Francisco-based restaurant," they said,
"doesn't charge $10".

Output: "␣The␣San␣Francisco-based␣restaurant␣,␣"␣they␣said␣,␣
"␣does␣n't␣charge␣$␣10␣"␣.

- This standard separates clitics (e.g., *doesn't* → *does* + *n't*), keeps hyphenated words together, and separates punctuation as individual tokens.
- Since tokenization is performed before all other NLP tasks, it must be fast and efficient.
- Therefore, tokenization is usually implemented using deterministic algorithms based on regular expressions and finite state automata, such as the regexp_tokenize function in NLTK, which can also handle ambiguities involving apostrophes.

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     (?:[A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+?:(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(?:\.\d+)?%?  # currency, percentages, e.g. $12.40, 82%
...   | \.\.\.              # ellipsis
...   | [][.,;"'?():_`-]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Byte-Pair Encoding (BPE) for Tokenization

- Byte-Pair Encoding (BPE) is an alternative approach to tokenization.
- Instead of treating words or characters as tokens, BPE learns tokens automatically from data.
- Traditional word-based tokenization fails when unknown words appear.
- NLP systems are usually trained on one corpus and tested on another.
- If a word appears in the test data but not in the training data, the system cannot process it properly.

Example of the Problem

- Suppose the training corpus contains the words:
  - *low, new, newer*
- The word *lower* does not appear in training.
- A word-based tokenizer treats *lower* as an unknown word.

BPE Solves the Problem

- BPE breaks words into frequent subword units.
- These subword units are learned from the training data itself.
- The word *lower* can be represented as subwords like *low* + *er*.
- This allows the system to understand and process unseen words.

Advantages of BPE

- Reduces the out-of-vocabulary (OOV) problem.
- Works well for languages with rich morphology.
- Learns a compact and useful vocabulary automatically.
- Widely used in modern NLP systems, including neural language models.

Subword Tokenization

- To address the problem of unknown words, modern tokenizers make use of subword tokenization, in which tokens smaller than full words are automatically learned from data.
- These subwords may be arbitrary character sequences or meaningful linguistic units such as morphemes, which are the smallest meaning-bearing units of a language.
- For example, the word *unlikeliest* consists of the morphemes *un-*, *likely*, and *-est*. In most modern tokenization schemes, the majority of tokens are complete words, but frequently occurring subwords like *-er* or *-est* are also included.

Representation of Unseen Words

- An important advantage of subword tokenization is its ability to represent unseen or unknown words.
- Any new word can be broken into a sequence of known subword units.
- For instance, the word *lower* can be represented as *low* and *er*, or, if necessary, as a sequence of individual characters.
- This greatly reduces the out-of-vocabulary problem in NLP systems.

Components of Subword Tokenization

- Most subword tokenization approaches consist of two main components: a token learner and a token segmenter.
- The token learner analyzes a raw training corpus and automatically induces a vocabulary of tokens.
- The token segmenter then uses this learned vocabulary to segment raw test sentences into valid tokens.

Common Subword Tokenization Algorithms

- Several subword tokenization algorithms are widely used in NLP.
- These include Byte-Pair Encoding (BPE), Unigram Language Modeling, and WordPiece.
- In addition, the SentencePiece library provides implementations of BPE and Unigram language model–based tokenization methods.

Byte-Pair Encoding

- Byte-Pair Encoding (BPE) is one of the simplest and most widely used subword tokenization algorithms.
- It was introduced by Sennrich et al. (2016) and is commonly used in modern NLP systems.
- BPE automatically learns subword tokens from a training corpus based on frequency information.

Initial Vocabulary and Learning Process

- Start with a vocabulary of individual characters.
- Scan corpus to find the most frequent adjacent pair of symbols.
- Merge the pair into a new token and add it to the vocabulary.
- Replace all occurrences of the pair in the corpus with the new token.
- Repeat multiple times, creating longer character sequences.
- Parameter k controls the number of merges; final vocabulary = original characters + learned tokens.

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w
2 l o w e s t _	
6 n e w e r _	
3 w i d e r _	
2 n e w _	

Operation Within Word Boundaries

- BPE is usually applied within words**,** not across word boundaries.
- Therefore, the training corpus is first separated using whitespace, and each word is treated as a sequence of characters along with a special end-of-word symbol.
- Frequencies of these word forms are used during the learning process.

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2 l o w e s t _	
6 ne w er_	
3 w i d er_	
2 ne w _	

Example of BPE Merging

- When BPE is applied to a small training corpus, it first counts all adjacent character pairs.
- For example, the pair **e r** may be the most frequent because it appears often in words like *newer* and *wider*. This pair is merged into a single symbol **er**.
- In subsequent steps, other frequent pairs such as **er_**, **n e**, and so on are merged.
- Each merge reflects a frequently occurring pattern in the training data, gradually building meaningful subword units.

merge	current vocabulary
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

Tokenization of Test Data

- After the vocabulary is learned, a **token segmenter** is used to tokenize new sentences.
- The segmenter applies the learned merge rules to test data in the same order in which they were learned, using a greedy strategy.
- Importantly, frequencies in the test data do not affect tokenization.
- As a result, known words like *newer* may be tokenized as a single unit, while unseen words like *lower* can be segmented into subwords such as **low + er**, allowing the system to handle unknown words effectively.

```
function BYTE-PAIR ENCODING(strings C, number of merges k) returns vocab V

  V ← all unique characters in C          # initial set of tokens is characters
  for i = 1 to k do                       # merge tokens k times
    t_L, t_R ← Most frequent pair of adjacent tokens in C
    t_NEW ← t_L + t_R                     # make new token by concatenating
    V ← V + t_NEW                         # update the vocabulary
    Replace each occurrence of t_L, t_R in C with t_NEW   # and update the corpus
  return V
```

Figure 2.13 The token learner part of the BPE algorithm for taking a corpus broken up into individual characters or bytes, and learning a vocabulary by iteratively merging tokens. Figure adapted from Bostrom and Durrett (2020).

1.10.2 WORD NORMALIZATION

- Word normalization is the process of converting words or tokens into a standard or consistent form.
- It helps handle cases where the same word appears in different forms, such as *USA* and *US* or *uh-huh* and *uhhuh*.
- Useful in tasks like information retrieval and extraction to match all relevant documents, even with minor spelling differences.

Case Folding

- Case folding is a word normalization process where all words are converted to lowercase.
- It helps treat words like *Woodchuck* and *woodchuck* as the same, improving performance in tasks like information retrieval and speech recognition.
- However, it is avoided in tasks such as sentiment analysis and machine translation because capitalization can change meaning (e.g., **US** vs **us**).

Lemmatization and Morphological Normalization

- Treats different word forms as equivalent in NLP applications.
- Example: Searching *woodchucks* also finds *woodchuck*.
- Important for morphologically rich languages (e.g., Polish: *Warszawa*, *w Warszawie*, *do Warszawy*).
- **Lemmatization:** Reduces words to a common base or dictionary form.

1.10.3 SENTENCE SEGMENTATION IN TEXT PROCESSING

- Sentence segmentation divides continuous text into individual sentences.
- It is important for NLP tasks like parsing, information extraction, and machine translation.

Role of Punctuation

- Question marks and exclamation points clearly mark sentence boundaries.
- Periods are ambiguous as they can appear in abbreviations (e.g., Mr., Inc.).
- Sometimes a period can be both an abbreviation marker and a sentence end.
- Hence, sentence segmentation is often combined with word tokenization.

Methods for Sentence Tokenization

- Most systems use a binary classifier to decide if a period ends a sentence.
- Approaches include:
  - Rule-based methods
  - Machine learning techniques
- Abbreviation dictionaries help improve accuracy.

Simple Tokenization Steps

- Separate clear punctuation like ?, !, parentheses.
- Handle commas carefully (not split inside numbers).
- Process apostrophes and word endings.
- Use rules and abbreviation lists for periods.

Finite-State Implementation

- Rule-based tokenizers can be implemented using regular expressions.
- These can be converted into finite-state transducers (FSTs) for efficiency.

1.11 DETECTING AND CORRECTING SPELLING ERRORS

The detection and correction of spelling errors is an integral part of modern word-processors and search engines, and is also important in correcting errors in optical character recognition (OCR), the automatic *OCR* recognition of machine or hand-printed characters, and on-line handwriting recognition, the recognition of human printed or cursive handwriting as the user is writing.

We can distinguish three increasingly broader problems:

1. **non-word error detection:** detecting spelling errors that result in non-words, (like *graffe* for *giraffe*).
2. **isolated-word error correction:** correcting spelling errors that result in nonwords, for example correcting *graffe* to *giraffe*, but looking only at the word in isolation.
3. **context-dependent error detection and correction:** using the context to help detect and correct spelling errors even if they accidentally result in an actual word of English (**real-word errors**). This can happen *Realword errors* from typographical errors (insertion, deletion, transposition) which accidentally produce a real word (e.g., *there* for *three*), or because the writer substituted the wrong spelling of a homophone or near-homophone (e.g., *dessert* for *desert*, or *piece* for *peace*).

Detecting Non-Word Errors

- Non-word errors are detected by checking each word against a dictionary.
- Words not found in the dictionary are marked as spelling errors (e.g., *graffe*).
- **Early research (Peterson, 1986):**
  - Suggested keeping dictionaries small.
  - Large dictionaries may contain rare words that resemble common misspellings (e.g., *wont* vs. *won't*, *veery* vs. *very*).
- **Later research (Damerau & Mays, 1989):**
  - Larger dictionaries are more beneficial than harmful.
  - Rare words may hide some misspellings, but large dictionaries reduce false positives.
  - Important for probabilistic spell-correction systems, which consider word frequency.
  - Modern spell-checking systems rely on large dictionaries for better accuracy.

Finite-State Morphological Parsers

- Provide an effective technology for implementing large dictionaries.
- When a morphological parser is defined for a word:
  - A **Finite-State Transducer (FST)** functions as a word recognizer.
  - An FST can be converted into a more efficient **Finite-State Automaton (FSA)** by applying the projection operation.
- FST-based dictionaries are efficient and suitable for large-scale lexical recognition.

Advantages of FST Dictionaries

- Represent **productive morphology** (e.g., English inflections like -s, -ed).
- Recognize new and valid combinations of stems and inflections automatically.
- Adding a new stem allows all its inflected forms to be recognized without manual input.

- Especially powerful for **morphologically rich languages**, where a single stem can produce dozens or hundreds of word forms.

Spell-Checking and Error Correction

- FST dictionaries assist in detecting and correcting spelling errors.
- **Error correction for isolated words:**
  - Finds the most likely correct word for a misspelled word (e.g., *graffe* → *giraffe*, *graf*, *craft*, *grail*).
- **Choosing the best correction:**
  - Uses a **distance metric** to measure closeness between the misspelled word and candidate words.
  - Words closer in spelling are considered more likely corrections (*giraffe* is closer to *graffe* than *grail*).
- Advanced methods use **probability theory** for more accurate similarity modeling.
- **Basic underlying method:** non-probabilistic **minimum edit distance algorithm**.

1.12 MINIMUM EDIT DISTANCE

- **String distance:** Measures similarity between two strings; smaller distance → more similar.
- **Deciding closeness:** Determining which of two words is closer to a third word is a special case of string distance.
- **Minimum edit distance (Wagner & Fischer, 1974):**
  - Smallest number of edits (insertion, deletion, substitution) needed to transform one string into another.
- Many string similarity algorithms are based on this concept.

Example: Minimum Edit Distance

- Distance between *intention* and *execution* = 5 edits.
- **Alignment:** Pairs each character of one string with a character (or empty space) in the other.
- Operations are represented as symbols:
  - **d** = deletion
  - **s** = substitution
  - **i** = insertion

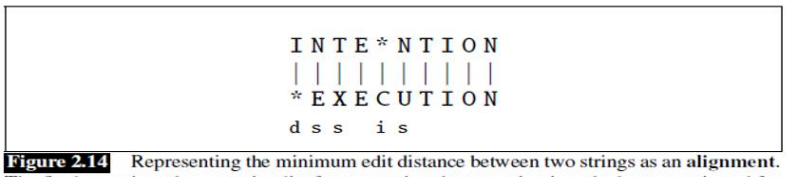


Figure 2.14 Representing the minimum edit distance between two strings as an **alignment**. The final row gives the operation list for converting the top string into the bottom string: d for deletion, s for substitution, i for insertion.

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. Thus the Levenshtein distance between *intention* and *execution* is 5.

Levenshtein also proposed an alternate version of his metric in which each insertion or deletion has a cost of one, and substitutions are not allowed. Using this version, the Levenshtein distance between *intention* and *execution* is 8.

The **minimum edit distance** is calculated using **dynamic programming**, a class of algorithms introduced by Bellman (1957). Dynamic programming solves problems by breaking them into smaller subproblems and combining their solutions in a table-driven manner. This approach is widely used in speech and language processing.

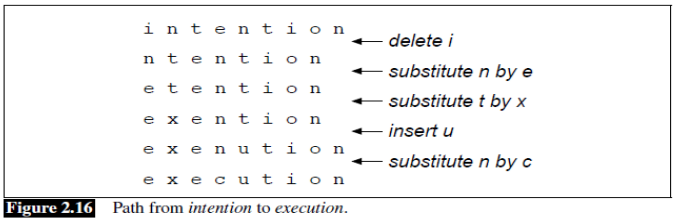


Figure 2.16 Path from *intention* to *execution*.

For example, to transform *intention* into *execution*, we follow a sequence of small edit operations.

- First, the letter i is **deleted** to form *ntention*.
- Next, n is **substituted** by e to get *etention*, followed by **substituting** t with x to form *exention*.
- Then, the letter u is **inserted**, producing *exenution*, and finally, n is **substituted** by c to arrive at *execution*.

Dynamic Programming and Minimum Edit Distance

- Breaks a complex problem into small, manageable subproblems.
- Combining all subproblems gives the overall solution efficiently.
- **Optimal substructure principle:**
  - If an intermediate string (e.g., *exention*) appears in the optimal path, the path to it must also be optimal.
  - A shorter path to any step would reduce total distance, contradicting optimality.
- This principle is the basis of dynamic programming.

Dynamic programming algorithms for sequence comparison compute similarity by constructing a **distance matrix**. The matrix has one row for each symbol of the **source sequence** and one column for each symbol of the **target sequence**, with the source along the side and the target along the bottom. In the case of minimum edit distance, this matrix is called the **edit-distance matrix**. Each cell, denoted as *edit-distance[i, j]*, represents the minimum distance between the first i characters of the target and the first j characters of the source. The value of each cell is calculated using a simple function based on neighboring cells. By starting from the initial cell and filling the matrix step by step, all distances can be computed. The value in each cell is obtained by taking the minimum of the three possible paths that reach that cell.

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(source[i]) \\ D[i,j-1] + \text{ins-cost}(target[j]) \\ D[i-1,j-1] + \text{sub-cost}(source[i], target[j]) \end{cases}$$

Minimum edit distance is useful not only for spelling correction but also for finding the minimum-cost alignment between two strings. String alignment is widely used in speech and language processing, such as computing word error rate in speech recognition and matching sentences in parallel corpora for machine translation.

$$D[i,j] = \min \begin{cases} D[i-1,j] + 1 \\ D[i,j-1] + 1 \\ D[i-1,j-1] + \begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases}$$

```
function MIN-EDIT-DISTANCE(source, target) returns min-distance

  n ← LENGTH(source)
  m ← LENGTH(target)
  Create a distance matrix D[n+1,m+1]

  # Initialization: the zeroth row and column is the distance from the empty string
  D[0,0] = 0
  for each row i from 1 to n do
    D[i,0] ← D[i-1,0] + del-cost(source[i])
  for each column j from 1 to m do
    D[0,j] ← D[0,j-1] + ins-cost(target[j])

  # Recurrence relation:
  for each row i from 1 to n do
      for each column j from 1 to m do
        D[i,j] ← MIN( D[i−1,j] + del-cost(source[i]),
                      D[i−1,j−1] + sub-cost(source[i],target[j]),
                      D[i,j−1] + ins-cost(target[j]))
  # Termination
  return D[n,m]
```

Figure 2.17 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g., $\forall x$, ins-cost$(x) = 1$) or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e., sub-cost$(x,x) = 0$).

Src\Tar	#	e	x	e	c	u	t	i	o	n
#	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	6	7	8
n	2	3	4	5	6	7	8	7	8	7
t	3	4	5	6	7	8	7	8	9	8
e	4	3	4	5	6	7	8	9	10	9
n	5	4	5	6	7	8	9	10	11	10
t	6	5	6	7	8	9	8	9	10	11
i	7	6	7	8	9	10	9	8	9	10
o	8	7	8	9	10	11	10	9	8	9
n	9	8	9	10	11	12	11	10	9	8

Figure 2.18 Computation of minimum edit distance between *intention* and *execution* with the algorithm of Fig. 2.17, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions.

To produce an alignment using the edit distance algorithm, the alignment can be viewed as a path through the edit-distance matrix. Each selected cell represents the alignment of characters from the two strings. Moving within the same row indicates an insertion, while moving within the same column indicates a deletion.

Figure 3.27 shows how alignment is computed using two steps. First, backpointers are stored in each cell of the edit-distance matrix to record where the minimum cost came from. Second, a backtrace is performed from the final cell to the initial cell by following these pointers, producing a minimum-distance alignment. Several public tools are available to compute edit distance, such as UNIX **diff** and the **NIST sclite** program.

n	9	↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙←↓ 12	↓ 11	↓ 10	↓ 9	↙ **8**
o	8	↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↓ 10	↓ 9	↙ **8**	← 9
i	7	↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9	↙ **8**	← 9	← 10
t	6	↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙ **8**	← 9	← 10	←↓ 11
n	5	↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ **8**	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙↓ 10
e	4	↙ 3	← 4	↙← **5**	← **6**	← 7	←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9
t	3	↙←↓ 4	↙←↓ **5**	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙ 7	←↓ 8	↙←↓ 9	↓ 8
n	2	↙←↓ **3**	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↓ 7	↙←↓ 8	↙ 7
i	**1**	↙←↓ 2	↙←↓ 3	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙ 6	← 7	← 8
#	**0**	1	2	3	4	5	6	7	8	9
	#	**e**	**x**	**e**	**c**	**u**	**t**	**i**	**o**	**n**

Figure 3.27 When entering a value in each cell, we mark which of the 3 neighboring cells we came from with up to three arrows. After the table is full we compute an **alignment** (minimum edit path) via a **backtrace**, starting at the **8** in the upper right corner and following the arrows. The sequence of dark grey cell represents one possible minimum cost alignment between the two strings.

VISION: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.