Natural Language Processing

Christian Rossi

Academic Year 2024-2025

Abstract

This course introduces students to the challenges and methodologies related to the analysis and production of natural language sentences, both written and spoken. The course explores the current role of stochastic models and Deep Learning, as well as new opportunities to combine traditional, formally based models with stochastic models. Topics covered include morphology, syntax, semantics, pragmatics, voice, prosody, discourse, dialogue, and sentiment analysis.

The course includes practical exercises where students can test the models and techniques presented in the lectures. Applications explored during the course will include: human-machine and human-human interaction analysis based on language; linguistic and prosodic analysis and generation for rehabilitation; pattern recognition and research for sentiment analysis in critical interactions; complexity analysis in text and overall spoken communication; and the development of user profiles that account for expression preferences in forensic, educational, and clinical contexts.

Contents

CHAPTER 1

Introduction

1.1 Natural language

The origins of spoken language are widely debated. Estimates range from as early as 2.5 million years ago to as recent as 60,000 years ago, depending on how one defines human language.

The development of written language, however, is more clearly documented. The first known writing systems emerged in Mesopotamia (modern-day Iraq) around 3500 BCE. Initially, these were simple pictograms representing objects, but over time, they evolved into abstract symbols representing sounds, paving the way for more complex communication.

The characteristics of human language are as follows:

- *Compositional*: language allows us to form sentences with subjects, verbs, and objects, providing an infinite capacity for expressing new ideas.
- *Referential*: we can describe objects, their locations, and their actions with precision.
- *Temporal*: language enables us to convey time, distinguishing between past, present, and future events.
- *Diverse*: thousands of languages are spoken worldwide, each with unique structures and expressions.

1.1.1 Natural Language Processing

One reason to care about Natural Language Processing (NLP) is the sheer volume of human knowledge now available in machine-readable text. With the rise of conversational agents, human-computer interactions increasingly rely on language understanding. Furthermore, much of our daily communication is now mediated by digital platforms, making NLP more relevant than ever.

However, NLP is a challenging field. Human language is highly expressive, allowing people to articulate virtually anything. Resolving this ambiguity is one of the core difficulties in computational linguistics. Moreover, meaning can be influenced by pronunciation, emphasis, and context, making interpretation even more complex. Fortunately, language is often redundant, allowing for error correction and inference even when mistakes occur.

1.1.2 History

The field of NLP has its roots in linguistics, computer science, speech recognition, and psychology. Over time, it has evolved through various paradigms, driven by advancements in formal language theory, probabilistic models, and Machine Learning (ML).

During World War II, early work in NLP was influenced by information theory, probabilistic algorithms for speech, and the development of finite state automata.

Between 1957 and 1970, two primary approaches emerged. The symbolic approach, based on formal language theory and AI logic theories, focused on rule-based processing. Meanwhile, the stochastic approach leveraged Bayesian methods, leading to the development of early Optical Character Recognition (OCR) systems.

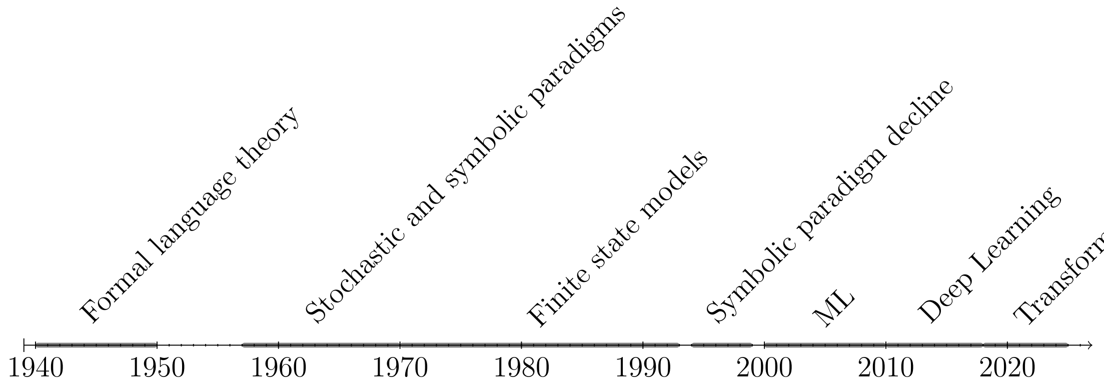
From 1970 to 1993, the focus shifted toward empirical methods and finite-state models. Researchers worked on understanding semantics, discourse modeling, and structural analysis of language.

By the mid-1990s, symbolic approaches began to decline, and the late 1990s saw a surge in data-driven methods, fueled by the rise of the internet and new application areas.

The 2000s marked a deep integration of ML into NLP. The increasing availability of annotated datasets, collaboration with ML and high-performance computing communities, and the rise of unsupervised systems solidified empiricism as the dominant paradigm.

From 2010 to 2018, ML became ubiquitous in NLP, with Neural Networks driving major advances in conversational agents, sentiment analysis, and language understanding.

Since 2018, the field has been revolutionized by Transformer architectures. Pre-trained language models, such as BERT and GPT, have enabled transfer learning at an unprecedented scale, leading to the rise of massive online language models.



1.2 Text analysis

Before applying any NLP algorithm, it is important to standardize and clean the text. Preprocessing ensures consistency and improves the accuracy of downstream tasks. Common cleaning steps include:

- Before tokenization, removing non-content information such as HTML tags, converting text to lowercase, and eliminating punctuation.
- After tokenization, filtering out stop-words (extremely common words that add little meaning), removing low-frequency words, and applying stemming or lemmatization to reduce vocabulary size.

1.2.1 Text mining

Text may need to be extracted from various sources, each with its own challenges:

- Textual documents, HTML pages, and emails often contain formatting elements that should be removed.

- Binary documents are more complex to process. In PDFs, text may be laid out in multiple columns, requiring careful reconstruction. If all PDFs follow a consistent format, handwritten rules may suffice; otherwise, ML techniques may be needed.

- Scanned documents require Optical Character Recognition, which relies on Deep Learning (DL) to convert images to text. However, Optical Character Recognition is not flawless and can introduce recognition errors.

1.2.2 Characters encoding

When storing and processing text, different character encodings must be considered.

ASCII encoding represents only 128 characters, mapping letters and symbols to numerical values. While sufficient for basic English text, it cannot handle many linguistic symbols.

UTF-8 encoding supports over 149,000 Unicode characters, covering more than 160 languages. Unicode is essential for processing texts that use non-Latin scripts. It also preserves special characters, such as diacritical marks in Italian and English.

1.2.3 Tokens

In many languages, spaces serve as natural boundaries between words, making tokenization straightforward. However, if spaces weren't available, we would need alternative methods to segment text. Since they do exist in most languages, they are commonly used for tokenization.

Despite this, tokenizing text isn't always simple. Hyphenated words can pose challenges, as some languages construct long, compound words that may need to be split for effective processing. In other cases, meaningful units are spread across multiple non-hyphenated words in multi-word expressions. Additionally, punctuation cannot always be blindly removed, as certain clitics (words that don't stand alone) depend on them for meaning.

For languages like Chinese, tokenization is even more complex since it does not use spaces to separate words. Deciding what constitutes a word is non-trivial, and a common approach is to treat each character as an individual token. Other languages, such as Thai and Japanese, require sophisticated segmentation techniques beyond simple whitespace or character-based tokenization.

A more advanced method, sub-word tokenization, can be useful for handling long words and capturing morphological patterns within a language. Instead of relying purely on spaces, data-driven techniques determine the optimal way to segment text. This is particularly important for ML applications, where models benefit from explicit knowledge of a language's structure. A common approach is byte-pair encoding.

In some tasks, text must be split into sentences rather than just words. Sentence segmentation often relies on punctuation marks, which typically indicate sentence boundaries. However, periods are more ambiguous, as they also appear in abbreviations, numbers, and initials. A common approach is to tokenize first and then use rule-based or ML models to classify periods as either part of a word or a sentence boundary.

1.2.4 Text normalization

In many applications, such as web search, all letters are converted to lowercase. This process significantly reduces the vocabulary size and improves recall by ensuring that variations in capitalization do not affect search results. Since users often type queries in lowercase, this normalization helps retrieve more relevant documents.

For classification tasks, removing case can simplify the learning process by reducing the number of distinct tokens. With fewer parameters to learn, models can generalize better even with limited training data.

However, case folding is not always beneficial. In some contexts, capitalization carries meaningful information. Machine translation and information extraction may also benefit from preserving case distinctions.

Beyond case folding, word normalization involves converting words or tokens into a standard format, ensuring consistency in text processing. This step is particularly crucial in applications like web search, where variations in word forms should not hinder retrieval performance.

Stop-words Stop-words are the most frequently occurring words in a language. They typically have extremely high document frequency scores but carry little discriminative power, meaning they do not contribute much to understanding the main topic of a text. Removing stop-words can sometimes improve the performance of retrieval and classification models, mainly by reducing computational and memory overhead. Eliminating common words can also speed up indexing by preventing the creation of excessively long posting lists. However, stop-word removal is not always beneficial. In some cases, stop-words play an important role in understanding meaning and context.

1.2.5 Morphology and lemmatization

Morphology, a fundamental concept in linguistics, refers to the analysis of word structure. At its core, it involves breaking words down into their smallest meaningful units, known as morphemes.

Definition (*Morpheme*)**.** A morpheme is the smallest linguistic unit that carries meaning.

A morpheme can be a root (base form) or an affix, which can appear as a prefix, infix, or suffix.

Definition (*Lexeme*)**.** A lexeme is unit of lexical meaning that exists regardless of inflectional endings or variations.

Definition (*Lemma*)**.** A lemma is the canonical form of a lexeme.

Definition (*Lexicon*)**.** A lexicon is the set of all lexemes in a language.

Definition (*Word*)**.** A word is an inflected form of a lexeme.

Lemmatization Lemmatization is the process of reducing words to their lemma, or base form. By normalizing words to a common root, it helps deal with complex morphology, which is essential for many languages with rich inflectional systems.

Stemming Stemming is a simpler approach that removes affixes based on predefined rules, often without considering the actual meaning or structure of the word. Unlike lemmatization, stemming does not require a lexicon.

Porter stemming algorithm (1980) is one of the most widely used stemming algorithms, it applies a set of rewriting rules to reduce words to their stems. While computationally efficient, stemming can introduce errors such as collisions (different words may be reduced to the same stem) and over-stemming (some words may be shortened excessively, losing meaning).

While stemming is computationally cheaper, lemmatization provides more linguistically accurate results, making it preferable for tasks requiring precise language understanding.

1.3 Regular expressions

Text documents are fundamentally just sequences of characters. Regular expressions provide a powerful way to search within these sequences by defining patterns that match specific character sequences. Regular expressions are useful for:

- *Pattern detection*: determine whether a specific pattern exists within a document.
- *Information extraction*: locate and extract relevant information from a document whenever the pattern appears.

Name	**Formula**	**Description**
Exact match	`aaa`	Matches the exact sequence `aaa`
Sequence choice	`(aaa|bbb)`	Matches either `aaa` or `bbb`
Wildcard	`.`	Matches any single character except for a newline
Character choice	`[]`	Matches any one character inside the square brackets
Newline	`\n`	Represents a newline character
Tab	`\t`	Represents a tab character
Whitespace	`\s`	Matches any whitespace character
Non-whitespace	`\S`	Matches any non-whitespace character
Digit	`\d`	Matches any digit (`[0-9]`).
Word character	`\w`	Matches any word character (`[a-zA-Z0-9]`)
Zero or more times	`*`	Matches the preceding character zero or more times
One or more times	`+`	Matches the preceding character one or more times
Zero or one time	`?`	Matches the preceding character zero or one time
Exactly n times	`{n}`	Matches n occurrences of the preceding character
From n to m times	`{n,m}`	Matches between n and m occurrences of the preceding character

1.3.1 Regular expressions in text mining

Regular expressions offer a powerful way to define patterns that can extract specific content from text documents. This allows for highly customizable and efficient text processing.

The advantages of regular expression based text extraction are:

- *Simplicity*: regular expressions are a straightforward way to specify patterns.
- *Precision*: extraction rules can be finely tuned to target specific patterns, which reduces false positives.

The limitations of regular expression based text extraction are:

- *Manual rule creation*: writing extraction rules usually requires manual effort, which can be time-consuming and complex.
- *False positives*: regular expressions can still yield false positives, where the pattern matches unintended content.
- *False negatives*: false negatives occur when the rules are not broad enough to capture all valid cases.
- *Lack of context awareness*: regular expressions typically work on isolated patterns, without understanding the context in which the pattern appears.

CHAPTER 2

Text classification

2.1 Supervised Learning

ML involves techniques that help machines become more intelligent by learning from past data to predict future outcomes.

Definition (*Machine Learning*)**.** A computer program is considered to learn from experience E when it improves its performance on a specific task T based on that experience, as measured by a performance metric P.

In Supervised Learning, each training example is represented as a vector in a feature space. These training examples are labeled with the correct class. The task is to divide the feature space in such a way that the model can make accurate predictions for new, unseen data points.

In practice, however, data is rarely perfectly clean or neatly separable. Often, different classes of data overlap, meaning they may not be linearly separable. Additionally, instances are described by many features, and not all features are equally useful for distinguishing between classes. Some features might provide more meaningful information, while others might be less relevant.

To address this, classifiers divide the feature space into regions, and the boundaries between these regions can either be linear or non-linear. Linear models use simple decision boundaries to separate classes. On the other hand, non-linear models are capable of creating more complex decision boundaries to better fit the data.

Training The process of training a model involves finding a formula that can predict the correct labels for new instances. The learning algorithm takes in the training data and their corresponding labels, and then searches for the best parameters for the model. These parameters are adjusted in order to minimize prediction loss on the training data. The learning algorithm operates based on its own settings, called hyperparameters, which control aspects of the learning process.

Hyperparameters Hyperparameters play a crucial role in determining the model's behavior. These are parameters that govern the learning algorithm itself, and they can influence the complexity of the model.

2.1.1 Overfitting

Overfitting is a common challenge in ML. As the model becomes more complex, the error on the training data tends to decrease. However, at some point, the model may begin to memorize the training data rather than learning generalizable patterns, leading to poor performance on unseen data. This is known as overfitting. The goal is to find the model that strikes the right balance, one that generalizes well to new, unseen data, rather than simply fitting the training data perfectly.

To prevent overfitting, we need to carefully select hyperparameters that control the model's complexity. Since the training data alone doesn't tell us how well the model will generalize, and the test data should be reserved for final evaluation, we use a separate validation dataset. This dataset is a portion of the training data held out during the training process. The model is trained multiple times with different hyperparameter settings, and its performance is evaluated on the validation set. By comparing how well each configuration generalizes, we can choose the hyperparameters that lead to the best performance.

2.2 Text classification

Text classification involves training a model to assign documents to specific categories. It's a widely-used task across various fields. Classification problems can take different forms, including:

- *Binary classification*: where the output is either one of two possible labels.
- *Ordinal regression*: where the output is an ordered value, representing categories with a natural rank.
- *Multi-class classification*: where the output corresponds to one category from a set of predefined options.
- *Multi-label classification*: where the output is a set of categories that can overlap or be chosen simultaneously.

2.2.1 Feature extraction

Text can be arbitrarily long, so it can't be fed directly into a model. To make text usable for ML, we first need to extract meaningful features. Features are the useful signals in the document that help predict its category. To do this, we need to convert text data into a vector of features that a classifier can process.

When training data is limited (with only a few documents available), some approaches to feature extraction include:

- *Syntax-based features*: such as the number of capitalized words.
- *Part-of-speech features*: like the count of verbs versus proper nouns.
- *Reading difficulty features*: such as average word or sentence length

However, the most common and effective features to extract are the words themselves. The vocabulary of the document provides significant signals. The frequency of word occurrences offers additional context.

One popular method is the Bag-of-Words (BoW) model. This model represents documents as vectors of word counts. It results in a sparse representation (long vectors with many zero entries).

One-hot encoding One-hot encoding could be an option to create fixed-dimensional feature vectors. However, there are practical limitations: represents each word as a binary feature, creating a vector where each dimension corresponds to a word in the vocabulary. To solve this, we often sum the one-hot encodings. This reduces the number of features to the size of the vocabulary. While this discards word order, it retains the critical information about the vocabulary and word occurrences.

2.2.2 Word frequencies

In text data, certain statistical laws describe how term frequencies behave across documents and collections:

- *Heap's law*: this law states that the vocabulary size grows with the square root of the document or collection length:

  $$V(l) \propto l^\beta$$

  Here, $\beta \approx 0.5$. This means the number of unique words in a document or collection increases slowly as the length of the document or the size of the collection grows.

- *Zipf's law*: this law describes the frequency of a token being inversely proportional to its rank:

  $$\mathrm{ctf}_t \propto \frac{1}{\mathrm{rank}(t)^s}$$

  Here, $s \approx 1$. In simple terms, a small number of words (the most frequent ones) appear very often, while a large number of words appear very rarely.

Heap's law is derived from Zipf's law and can be understood through models like random typing, showing how the vocabulary of a document or collection grows more slowly compared to its length.

Bag of Words The Bag of Words model represents a document as a collection of its terms, ignoring grammar and word order but keeping track of the frequency of each word. In this model:

- The vocabulary of a document is much smaller than the vocabulary of the entire collection, so the terms in the document generally give a good representation of its content.

- The BoW representation typically includes the count of occurrences of each term, although a binary representation (indicating presence or absence of words) can also be used with minimal loss of information.

However, the BoW model has limitations. It produces a sparse representation of the text, meaning most of the values in the vector are zero. The model completely ignores word order, which means it cannot capture the sequence or context in which words appear.

To improve this, we can extend BoW to include n-grams, which capture sequences of words. This can enhance performance, but it significantly increases the number of features, requiring more data to avoid overfitting.

2.2.3 Preprocessing

In NLP, preprocessing is a critical step to prepare the text data for ML. Common preprocessing tasks include tokenization, spelling correction, and other forms of cleaning the text.

Spelling correction When dealing with text data, misspellings can often occur, especially with user-generated content. Correcting these misspellings can improve model performance. One way to approach spelling correction is by using a probabilistic model.

If we had an enormous corpus of misspellings and their correct versions, we could estimate the probability of a misspelling being corrected in a certain way by using string edit distance, which measures how much one string differs from another. The edit distance counts the minimum number of operations (insertions, deletions, substitutions, or transpositions) needed to convert one string into another.

To apply this to spelling correction, we use Bayes' Rule to reverse the conditional probability:

- The likelihood of a misspelling being corrected to a particular word can be computed using the edit distance between the misspelled word and the candidate correction.
- The prior Pr(correct) represents the popularity or frequency of the word in a large corpus. This can be estimated by counting how often the candidate correction appears in a corpus.
- The likelihood Pr(observed | correct) represents the probability of observing the misspelled word given the correct word.

In practical terms, the prior shows how often a word appears in a large corpus, while the likelihood shows how likely it is that the observed misspelling corresponds to a particular correction, based on string edit distance. Since the denominator in Bayes' Rule is the same for all candidate corrections, we can ignore it during the calculation and normalize probabilities later.

To improve spelling correction, we can incorporate contextual information. This can be done by considering bi-grams (pairs of consecutive words) instead of just individual words (uni-grams). By calculating the bi-gram probabilities Pr(bigram), the model can use both the misspelled word and the previous word in the sentence as features. This approach turns the spelling correction process into a Naïve Bayes model with two features: the misspelled word and the previous word in the sentence.

2.3 Linear classifier

In text classification, where documents are often represented with a bag-of-words model, linear classifiers are commonly used due to the high dimensionality of the feature space. Linear models work by assigning a parameter to each word in the vocabulary, making them highly interpretable. This approach allows us to understand which terms have the greatest impact on the prediction and the extent of their influence.

Decision boundaries Linear classifiers create decision boundaries that are represented as hyperplanes in an n-dimensional vector space. The model includes a weight vector, which is the same size as the feature vector, along with a bias term.

2.3.1 Multinomial Naïve Bayes

Naïve Bayes is one of the oldest and simplest text classification algorithms. It is called naïve because it makes a simplifying assumption: that word occurrences are statistically independent of each other given the class label.

This assumption means that each word contributes independent information about the class. It simplifies the process of calculating the model's parameters, making the algorithm easy to implement. However, in practice, this assumption doesn't hold since words are often correlated with each other. Despite this, Naïve Bayes still produces effective predictions, though the assumption can make the model seem overly confident in certain cases.

If all instances of a word appear exclusively in one class, we can have issues with probability estimation. To avoid this problem, we use smoothing, which consists in adding a small pseudo-count α for each feature. This helps prevent zero probabilities for unseen word-class combinations. The value of α can be selected to optimize performance or set to a default value (if $\alpha = 1$, this technique is known as Laplace smoothing).

Independence assumption The independence assumption is not necessarily a big problem. the assumption simplifies both model estimation and prediction, which, in turn, makes the process more efficient. While this theoretically reduces the model's accuracy slightly, in practice, Naïve Bayes often works well for some tasks.

Advantages	
Speed	Naïve Bayes is incredibly fast to train, requiring just one pass over the training data. No need for complex optimization routines like gradient descent
Stability	It's a reliable model even with limited data. If the conditional independence assumption holds, it provides the best possible performance
Disadvantages	
Scalability	Naïve Bayes doesn't perform as well on large datasets compared to other classifiers because redundant features are counted multiple times
Calibrating probabilities	The predicted probabilities are not well calibrated, meaning they can be less reliable for certain applications

2.3.2 Logistic regression

The farther a point is from the decision boundary, the more confident we are in our prediction. The signed distance of a point from the hyperplane is given by:

$$s(\mathbf{x}) = \boldsymbol{\theta}\mathbf{x} - b$$

To convert the signed distance $s(x)$ into a probability, we need a function that maps the entire range of real values $\mathbb{R}$ to the probability range $[0, 1]$. The standard function used for this purpose is the logistic curve (also known as the sigmoid function):

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

This function outputs a probability of 0.5 at the decision boundary ($s = 0$). The slope, or the speed of probability change, depends on the magnitude of $\boldsymbol{\theta}$.

Advantages	
Well-calibrated probabilities	Logistic regression produces well-calibrated probability estimates
Scalability	It can be trained efficiently and scales well to large numbers of features
Interpretability	The model is explainable, since each feature's contribution to the final score is additive
Disadvantages	
Linearity assumption	Logistic regression assumes feature values are linearly related to log-odds
Sensitivity to assumptions	If the linearity assumption is strongly violated, the model will perform poorly

2.3.3 Support Vector Machines

Imagine a dataset where two classes are clearly separable into two groups. There are many possible positions for the linear decision boundary. We want to select a boundary that generalizes well to new, unseen data, avoiding overfitting.

The SVM approach finds the maximum margin hyperplane that separates the two classes. The margin, denoted γ, is the distance from the hyperplane to the closest points on either side. These closest points are called support vectors. Support vectors are the points that lie exactly on the margin. They prevent the margin from expanding and thus help define the location of the boundary. In a d-dimensional space, you need at least $d + 1$ support vectors to define the hyperplane.

In contrast to logistic regression, where the position of the hyperplane depends on the entire dataset, the SVM hyperplane's position is determined only by the closest points. The convex hull of the data points helps define the boundary, and moving internal points does not affect the hyperplane.

Hard margin SVM A basic SVM is also a linear classifier that finds a hyperplane in feature space that best separates the two classes. While logistic regression and Naïve Bayes also find linear decision boundaries, the difference lies in the loss function used to find the model parameters:

- Logistic regression uses negative log-likelihood, which penalizes points based on the probability of incorrect predictions, even if they are correctly classified.

- SVM uses hinge loss, which only penalizes points that are on the wrong side of the margin (or very close to it).

Mathematically, the loss function for SVM is:

$$\mathcal{L}(\mathbf{w}) = \sum_i w_i^2 + \sum_j \varepsilon_j$$

Here, ε_j is the error for a prediction (x_j, y_j), and it is defined as:

$$\varepsilon_j = \max(0, 1 - y_j \mathbf{w} x_j)$$

This formulation reflects the hinge loss, which penalizes misclassified points or those close to the margin.

Soft margin SVM In the case of non linearly separable dataset, SVMs still aim to separate the classes by penalizing points that are on the wrong side of the margin, based on their distance from the hyperplane. Support vectors are now the points that are either misclassified or very close to the margin, contributing a non-zero amount to the loss function.
The objective function to minimize remains similar to the hard margin case, but it includes a penalty for misclassified points:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2} \sum_i w_i^2 + C \sum_j \varepsilon_j$$

Here, ε_j is the distance from the j-th support vector to the margin and C is a hyperparameter that controls the trade-off between minimizing the margin size and penalizing errors. A large C places more emphasis on minimizing misclassifications, while a smaller C allows a larger margin even at the cost of more misclassifications.

SVM and logistic regression Both SVM and logistic regression are linear classifiers, but they differ in the loss functions they use: Logistic regression uses log-likelihood, which penalizes points based on the probability of incorrect predictions, including those correctly classified. SVM uses hinge loss, which only penalizes points on the wrong side of the margin or those very close to it.

2.4 Model Evaluation

When evaluating a classification model, a common starting point is the confusion matrix, which summarizes the true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Several metrics are derived from the confusion matrix to assess the performance of a model.

Accuracy Accuracy is the proportion of correct predictions, and it can be calculated as:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Precision Precision measures the proportion of positive predictions that were actually correct. It is defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

A high precision indicates that when the model predicts positive, it is likely to be correct.

Recall Recall, also known as sensitivity or true positive rate, measures the proportion of actual positives that were correctly identified by the model. It is calculated as:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

High recall means the model is good at identifying positive instances, though it might also include more false positives.

F-measure The F1 score combines precision and recall into a single metric by taking the harmonic mean of the two:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score is particularly useful when you need a balance between precision and recall, especially when the class distribution is imbalanced.

Area under the ROC curve The Area under the ROC Curve (AuC) is another important metric, often used when dealing with different thresholds for making predictions. It measures the area under the Receiver Operating Characteristic (ROC) curve, which plots the true positive rate against the false positive rate at various thresholds. A higher area under the curve indicates better model performance, as it signifies that the model is better at distinguishing between classes regardless of the threshold.

2.4.1 Multi-class classifiers

When working with multi-class classifiers, the confusion matrix expands to an $n \times n$ matrix, where n is the number of classes. In this case, precision and recall are calculated for each class, treating each class as the positive class in a one-vs-all fashion.

For a given class i, precision is the ratio of true positives for that class to the sum of true positives and false positives. Similarly, recall for class i is the ratio of true positives for that class to the sum of true positives and false negatives.

To combine the precision and recall across all classes, two methods are commonly used:

- *Macro-average*: takes the average of the precision and recall across all classes, treating each class equally, regardless of how many instances belong to that class. This method gives equal importance to all classes, which can be useful when the classes are imbalanced.
- *Micro-average*: aggregates the true positives, false positives, and false negatives across all classes before calculating precision and recall. This method gives more weight to classes with more data points. The micro-average is useful when the number of data points varies significantly between classes.

CHAPTER 3

Text search and clustering

3.1 Introduction

Information retrieval is the task of finding relevant content that match a user's information need. To achieve this, we typically extract keywords from the user's query and search for documents containing those keywords.

Vocabulary matching A simple but effective approach in text retrieval is vocabulary matching. If the vocabulary is well-distributed across documents, we can use fast indexing techniques to quickly locate relevant documents. However, this method has limitations. Some queries may not have an exact match in the document collection. Many documents might contain all the query terms, making it difficult to rank them effectively. To address these challenges, we can:

- Assign scores to keywords based on how discriminative they are.
- Expand document representations by incorporating additional signals, such as page importance.
- Train ML models to improve retrieval performance.

Classifier An alternative approach is to train a classifier that directly predicts the relevance of a document to a query. This involves representing both the query and the document as a combined feature vector and predicting the probability that a user finds the document relevant to the query. However, this approach presents several challenges:

- A simple linear classifier would fail to capture complex interactions between query and document terms.
- A non-linear model that accounts for pairwise term interactions would be extremely large.
- Training such a model would require massive labeled datasets of (query, document, relevance) pairs.
- Retrieval speed would suffer if we had to score every document individually.

3.2 Term weighting

In information retrieval, term weighting plays a crucial role in determining the relevance of documents to a query. The idea is to identify a subset of query terms that are most likely to return the most relevant documents. Generally, the smaller the subset of terms, the more specific and on-topic the resulting document set is likely to be. Thus, we rank documents based on how small the set of relevant documents is for a given query term subset.

One way to estimate how many documents a query term subset would return is by calculating the probability that a random document contains those terms. Documents can then be ranked based on how unlikely it is to see so many query terms in them.

Assuming the terms in the query are independent, we can express the probability of a document containing all the terms in the query as:

$$\Pr(q \subseteq d') = \prod_{t \in q} \Pr(t \in d') = \prod_{t \in q} \frac{\mathrm{df}_t}{N}$$

Here, df_t is the document frequency, or the number of documents containing term t and N is the total number of documents in the corpus.

3.2.1 Inverse Document Frequency

To rank documents, we can score them based on how unlikely it is for them to contain all the query terms. This gives rise to the inverse document frequency weighting:

$$\mathrm{score}(d) = -\log \prod_{t \in q \cap d} \Pr(t \in d') = \sum_{t \in q \cap d} \log \frac{N}{\mathrm{df}_t}$$

IDF is a measure that assigns weights to terms based on how rare they are across the entire document collection. This is a standard measure from information theory that quantifies the information gained from observing a term. Essentially, the IDF measures the surprise or information content of encountering a term in a document. This same concept is used in text compression algorithms to determine how many bits should be used to represent a word.

A common variation of IDF uses the odds of observing a term rather than the probability, which results in the following document score:

$$\mathrm{score}(d) = \sum_{t \in q} \frac{N - \mathrm{df}_t + 0.5}{\mathrm{df}_t + 0.5}$$

Here, a smoothing factor of 0.5 is added to all counts to prevent terms with very low frequencies from disproportionately affecting the ranking. This smoothing helps to handle rare terms without letting them dominate the results.

3.2.2 Term Frequency Inverse Document Frequency

While IDF helps to weight terms by their rarity, there's more to a document than just its vocabulary. Some documents may contain the same query term multiple times, making them more likely to be relevant to the query. To account for this, we introduce term frequency (the number of times a term appears in a document). The simplest way to include term frequency is to multiply the IDF score by the term frequency:

$$\mathrm{score}(q, d) = \sum_{t \in q} \mathrm{tf}_{t,d} \log \frac{N}{\mathrm{df}_t}$$

Here, $\mathrm{tf}_{t,d}$ is the number of occurrences of term t in document d.

This can be motivated as follows: instead of calculating the probability that a random document contains the term, we calculate the probability that a document contains the term exactly k times:

$$\Pr(t,k) \cong \Pr(\text{next token} = t)^k$$

The next token probability is estimated using the term occurrences across the entire collection:

$$\Pr(\text{next token} = t) = \frac{\mathrm{ctf}_t}{\sum_{t'} \mathrm{ctf}_{t'}}$$

Here, ctf_t is the collection term frequency for term t and $\sum_{t'} \mathrm{ctf}_{t'}$ is the total term frequency across all terms. The score can then be expressed as:

$$\mathrm{score}(q,d) = -\sum_{t \in q} \mathrm{tf}_{t,d} \log \frac{\mathrm{ctf}_t}{\sum_{t'} \mathrm{ctf}_{t'}}$$

Although this formulation is slightly different, it's conceptually similar to TF-IDF. Using document frequency instead of collection frequency doesn't drastically change the outcome, and in some cases, may even make the formula more robust.

Variations In practice, TF-IDF has been found to perform well, but it assumes a linear relationship between term frequency and document relevance. In most cases doubling the occurrences of a term in a document should not double the document's score. The score should improve with more occurrences of the term, but not linearly.

As a result, common alternatives include using a logarithmic scale for term frequency:

$$\log(1 + \mathrm{tf}_{t,d}) \qquad \max(0.1 + \log(\mathrm{tf}_{t,d}))$$

3.3 Text normalization

When ranking documents, it's important to normalize for document length. Longer documents tend to have a larger vocabulary, which makes it more likely they will contain the query terms. However, this doesn't necessarily mean they are more relevant to the user's search. In fact, shorter documents with the same term count should often be preferred.

3.3.1 Document length normalization

One simple way to normalize for document length is to divide the term frequency by the document length. However, the most common method of normalization uses the L2 norm (also called the Euclidean norm) instead of the L1 norm (which is simply dividing by the document length).

3.3.2 Cosine similarity normalization

In the Vector Space Model, each document is represented as a vector of term frequencies weighted by their inverse document frequency. The vector for a document might look like this:

$$\mathbf{d} = (\mathrm{tf}_{1,d} - \mathrm{idf}_1, \ldots, \mathrm{tf}_{1,d} - \mathrm{idf}_n)$$

To compute the similarity between a query and a document, we measure the cosine of the angle between their vectors. The cosine similarity formula is:

$$\text{similarity}(\mathbf{d}_1, \mathbf{d}_2) = \frac{\mathbf{d}_1 \cdot \mathbf{d}_2}{\|\mathbf{d}_1\| \, \|\mathbf{d}_2\|}$$

Here $\mathbf{d}_1$ represents the query vector, and $\mathbf{d}_2$ represents the document vector. The cosine of the angle is used because it produces a similarity value in the range $[0, 1]$. To calculate the cosine similarity, the vectors are normalized by their Euclidean (L2) norm, rather than the length of the document in terms of tokens (which would be an L1 norm).

There have been many studies into alternative methods of length normalization:

- *Pivoted Length Normalization* (PVL): aims to retain the beneficial information from longer documents while preventing them from being unfairly favored. The idea behind PVL is that longer documents generally contain more information, but simple length normalization could lose valuable length information. Instead, PLN adjusts the normalization to account for both the document length and the average document length in the corpus.

- *Best Match 25*: builds upon the ideas of TF-IDF and length normalization. The influence of a term on the document's score decreases as its frequency increases. There are parameters which allow fine-tuning based on the corpus.

3.4 Indices

Search engines are designed to deliver results as quickly as possible, since any delay can impact user experience and attention. Responses must be returned within tenths of a second, so search engines are optimized for speed.

At the heart of this efficiency is the inverted index, which is the core data structure used by search engines to retrieve documents.

3.4.1 Inverted indices

An inverted index consists of posting lists, which map term IDs to document IDs. The basic idea is to create a mapping between terms and the documents that contain them, so that when a user searches for a term, the system can quickly find all the relevant documents.

To optimize for speed and reduce storage space, inverted indices often use integer compression algorithms, allowing for quick decompression and reducing the overall size of the index.

When calculating a retrieval function, the process typically involves joining posting lists. The documents within these lists are sorted by term frequency. This sorting allows for early termination of the results list computation, so irrelevant documents are discarded quickly.

3.4.2 Positional indices

In many cases, it's not just about whether a term appears in a document, but where it appears. To capture this, some indices maintain positional information, recording the exact locations of terms within documents. This allows for the calculation of proximity between query terms, which can be a useful indicator of relevance.

Moreover, the location of words within a webpage can influence their importance. In addition, certain statistically significant bi-grams and trigrams are often identified and indexed separately. These are usually discovered using a technique like point-wise mutual information, which measures the association between terms. These bi-grams or trigrams often have their own posting lists, as they can provide more context to queries.

3.4.2.1 Crawlers

To populate the index, web crawlers scour the web, following hyperlinks to discover and add new pages to the search engine's database. Effective crawling involves two main challenges:

- *Prioritizing URLs*: the crawler must decide which URLs to visit first based on factors like relevance and likelihood of finding fresh content.
- *Re-visiting websites*: determining how often to revisit a website to check for updates is critical to ensuring that the index remains fresh and up to date.

At the scale of the web, crawlers must also be robust enough to handle different types of content, including dynamically generated pages. Additionally, web crawlers must detect and manage duplicate content. Many different URLs may lead to the same content, and the crawler needs to ensure that it doesn't index the same page multiple times.

To manage these challenges, a distributed crawler architecture is typically used, with a centralized URL list to keep track of the pages the crawler needs to visit. Crawlers also respect robots.txt files, which are placed in the root directory of websites. These files tell crawlers which pages or sections of the site they are allowed to crawl and index, helping website owners manage how their content is indexed.

3.5 Ranking

In web search, search engines rely on a variety of indicative features to determine the most relevant results for a user query. Search engines combine hundreds of signals to generate the most relevant search results. To do this efficiently, rank learning offers an automated and coherent method of combining diverse signals into a single retrieval score. It optimizes this score based on metrics that are important to users.

3.5.1 Re-ranking

The re-ranking process follows these steps:

1. *Start with a query*: the user enters a search query.
2. *Generate initial ranking*: use keyword-based ranking to retrieve an initial set of results.
3. *Truncate ranking*: limit the ranking to a manageable number of candidates for further evaluation.
4. *Calculate feature values*: compute relevant features for each candidate document.
5. *Normalize features*: normalize each feature at the query level to make the comparison between documents more consistent.
6. *Training*: use ground truth relevance labels to train the model.

3.5.2 Ranking metrics

There are several metrics used to evaluate the performance of search engine rankings. Here's a breakdown:

- *Precision at depth k*: the percentage of relevant documents in the top k results:

$$\text{precision} = \frac{\text{number of relevant documents in top } k}{k}$$

- *Recall at depth k*: the percentage of all relevant documents that are found in the top k results.

$$\text{recall} = \frac{\text{number of relevant documents in top } k}{\text{total relevant documents}}$$

- *F-measure at depth k*: a combination of precision and recall, providing a single score that balances both:

$$\text{F1} = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

- *Mean Average Precision*: this is the average of the precision values at each rank position where a relevant document appears. It estimates the area under the precision-recall curve:

$$\text{MAP} = \frac{\sum_{k=1}^{n}(\text{precision}(k) \cdot \text{rel}(k))}{\text{total number of relevant documents}}$$

- *Normalized Discounted Cumulative Gain*: this metric is more faithful to the user experience, as it gives lower ranks less importance, aligning with the natural way users interact with search results. It's normalized at the query level:

$$\text{NDCG}(Q, k) = \frac{1}{|Q|}\sum_{j=1}^{|Q|} Z_{kj} \sum_{m=1}^{k} \frac{2^{R(j,m)} - 1}{\log_2(1 + m)}$$

3.5.3 Regression re-ranking

Re-ranking can be treated as a regression problem, where the goal is to predict the relevance of a document based on various features. The model can then use standard regression techniques to combine these features and predict relevance scores. During training, the loss function can be defined in three different ways:

- *Point-wise*: this approach calculates the loss based on individual query-document pairs (Mean Squared Error).

- *Pairwise*: this method considers the relative ordering of document pairs (number of incorrectly ordered pairs).

- *List-wise*: this considers the entire ranking list (optimizing NDCG at the query level).

3.6 Text clustering

Text clustering is the process of grouping documents into coherent subgroups based on similarities in their content. These subgroups can be based on various criteria.

The key challenge in text clustering is to accurately measure the similarity between documents. This similarity is typically determined by analyzing the content of the documents. Traditionally, documents are represented as tf-idf vectors, which are scaled bag-of-words representations. In this approach:

- *Sparse representation*: most of the values in the vector are zero, meaning that each document is represented by a vector with many empty or zero entries.
- *Similarity measure*: the similarity between documents is usually based on the number of shared words, with the importance of each word being scaled by its rarity.

3.6.1 Clustering algorithms

There are several popular clustering algorithms used for grouping documents based on similarity.

3.6.1.1 k-means

k-means searches for exactly k clusters, each represented by a centroid, in the following way:

1. Randomly initialize k centroids.
2. Assign each data point to the nearest centroid.
3. Recompute the centroids by averaging the data points in each cluster.
4. Repeat steps 2 and 3 until convergence.

It scales well to large datasets and does not need pairwise distance calculations needed, which makes it faster. However, it assumes clusters are globular, which may not be ideal for text data, relies on the Euclidean distance metric, which might not be the best choice for all types of data.

The number of clusters must be specified in advance. Choosing the right value of k can be tricky, though the elbow method can help. The algorithm can converge on a local minimum; running it multiple times can help mitigate this issue. Scaling of the data affects clustering results, especially for text, where tf-idf weighting and document length normalization are important.

3.6.1.2 k-medoids

k-medoids is similar to k-means, but it uses medoids instead of centroids. A medoid is a point from the dataset itself that is closest to all other points in the cluster. In each iteration, the algorithm reassigns data points to the cluster with the closest medoid and then recomputes the medoids.

It is flexible because it works with various distance metrics. Since a medoid is an actual data point (not a mean), it provides a more realistic representation of the cluster. However, it has higher computational complexity compared to k-means because it requires calculating distances between pairs of points, which is an $\mathcal{O}(n^2)$ operations.

3.6.1.3 Agglomerative hierarchical clustering

Agglomerative hierarchical clustering is a hierarchical clustering builds a hierarchy of clusters, known as a dendrogram. Agglomerative hierarchical clustering works by merging smaller clusters bottom-up in the following way:

1. Assign each document to its own group.
2. Merge the two most similar groups.
3. Repeat until only one group remains.

To merge clusters, hierarchical clustering computes distances between them using various linkage criteria:

- *Complete-linkage*: maximum distance between points in two groups.
- *Single-linkage*: minimum distance across groups.
- *Average-linkage*: average distance across groups.

The choice of linkage criteria affects the shape and tightness of the clusters. Complete or average-linkage tends to create tight, globular clusters. Single-linkage can lead to long, thin clusters.

One advantage is that it works with any distance metric or similarity function. Thus, the dendrogram provides useful insight into the structure of the data. However, it has an high time complexity makes it unsuitable for large datasets.

3.6.1.4 Density-Based Spatial Clustering of Applications with Noise

DBScan is a density-based clustering algorithm that does not require the number of clusters to be specified in advance. It has two key parameters, namely ε (radius of the neighborhood around each point) and minPoints (minimum number of points required to form a cluster). The algorithm classifies points as:

- *Core points*: points with at least minPoints within their ε-neighborhood.
- *Border points*: points that are not core points but lie within the ε-neighborhood of a core point.
- *Noise points*: points that are neither core nor border points.

In this algorithm we do not need need to specify the number of clusters upfront, it is robust to noise and outliers, and can find arbitrary-shaped clusters, making it highly flexible. However, the performance depends on the chosen parameters and may not work well when clusters have different densities.

3.6.1.5 Topic modelling

Topic modeling is a soft clustering technique for documents, meaning that each document can belong to multiple clusters (or topics) to varying degrees. It is a way to extract the underlying themes or topics from a collection of documents.

Topic modeling reduces the dimensionality of documents by representing them in a lower-dimensional space compared to the high-dimensional vocabulary space. Each topic is described by a probability distribution over words, where the terms should ideally reflect a single theme. Similarly, documents are represented as probability distributions over topics.

Matrix decomposition Topic modeling is often approached as a form of matrix decomposition. The general idea is to decompose a term-document matrix (which represents word counts or tf-idf scores for each term in each document) into smaller matrices representing terms · topics and topics · documents. Instead of dealing with a large $V \times D$ matrix (where V is the vocabulary size and D is the number of documents), we decompose it into two much smaller matrices:

$$V \times T \quad \text{(terms} \cdot \text{topics)} \qquad T \times D \quad \text{(topics} \cdot \text{documents)}$$

Here, V is the vocabulary size, T the number of topics, and D the documents count. This decomposition reduces the number of parameters that need to be estimated, making the topic modeling process more efficient. The most used modeling techniques are:

- *Latent Dirichlet Allocation*: LDA is the most famous technique in topic modeling. It uses a Dirichlet prior to estimate the parameters. In LDA, each document is assumed to be a mixture of topics, and each topic is a mixture of words.
- *Non-negative Matrix Factorization*: a related technique to LDA, which factorizes the document-term matrix into two non-negative matrices representing term-to-topic and topic-to-document relations.
- *Latent Semantic Indexing*: LSI applies Singular Value Decomposition to a TF-IDF matrix to uncover latent semantic structure.

Applications Topic modeling helps address several challenges:

- *Polysemy*: it can disambiguate words with multiple meanings.
- *Synonymy*: it identifies synonyms that might be used interchangeably.
- *Short documents*: it improves the representation of documents that may have limited vocabulary due to their short length.

In addition to its utility in improving document representation, topic modeling is also useful for dimensionality reduction. After modeling topics, further dimensionality reduction techniques can help visualize collections of documents in a more interpretable way.

3.6.1.6 Generative model

The Generative Model for Latent Dirichlet Allocation is a probabilistic process that describes how the words in a document are generated from topics. Here's how it works:

1. *Choose word proportions for each topic*: each topic is associated with a probability distribution over words.
2. *Choose topic proportions for each document*: each document is modeled as a distribution over topics.
3. *For each word in a document*: choose a topic based on the document's topic proportions and choose a word based on the topic's word distribution.

Estimating the parameters of the topic model involves updating the topic and word distributions iteratively. This is typically done using Bayesian priors to avoid overfitting, ensuring that the model doesn't just memorize the data but rather generalizes well. Gibbs sampling or other sampling techniques are used to avoid local maxima during parameter optimization.

The hyperparameters of this method are:

- α: the prior on the topic distribution for each document (controls how concentrated the topic distribution is).
- β: the prior on the word distribution for each topic (controls how concentrated the word distribution is).

By iteratively updating these parameters and sampling from the distribution, the model learns the underlying structure of topics in a collection of documents.

CHAPTER 4

Language models

4.1 Introduction

Definition (*Statistical language model*)**.** A statistical language model is a probability distribution over sequences of words.

Given this distribution over sequences, we can condition the next word based on previous words and generate new sequences by sampling from it. In essence, language models serve as general-purpose text generators.

Language models identify statistical patterns in text and leverage these patterns to predict the next word in a sequence. By predicting each subsequent word with increasing accuracy, language models can generate entire sentences or even longer passages.

4.1.1 Markov models

Natural language utterances can be of arbitrary length, but we only have a finite set of parameters to model them. One way to define a probability distribution over such variable-length sequences is to predict the next word based on a fixed number of previous words.

The simplest models for this are n-gram models, which count sequences of n words in a large corpus. Using longer n-grams provides better predictions, though the model can become more sparse and complex as n increases.
Several techniques improve the performance of Markov models:

- *Smoothing* (regularization): this technique adds a small constant to all counts before estimating probabilities, which helps avoid assigning zero probability to unseen n-grams. The smoothed probability can be computed as:

  $$\Pr(w_n \mid w_{n-1}, w_{n-2}) = \frac{\text{count}(w_{n-2}w_{n-1}w_n) + \alpha}{\text{count}(w_{n-2}w_{n-1}) + V\alpha}$$

  Here, α is a pseudocount (often set to $\alpha = 1$), and V is the size of the vocabulary. This ensures that all possible n-grams have a non-zero probability, even if they haven't been seen in the training data.

- *Backoff*: instead of inventing values for unseen n-grams, the backoff technique uses lower-order models when an n-gram is not found in the training data. This helps maintain the flow of predictions while keeping the model manageable.

- *Interpolation*: this technique combines higher-order and lower-order models by blending their probabilities. To determine the weight of each model, interpolation parameters (lambdas) are chosen to maximize the likelihood on a held-out development set.

Generative Markov model A generative Markov model can be used to estimate the probability of the next word and generate text in various ways:

- *Greedy*: this approach chooses the most probable term:
$$w^* = \operatorname*{argmax}_t \Pr(w_n = t \mid w_{n-k}, \dots, w_{n-1})$$
- *Random sampling*: in this method, a term is sampled according to its probability:
$$w^* \sim \Pr(w_n \mid w_{n-k}, \dots, w_{n-1})$$
- *Top-k sampling*: sampling is restricted to the top k most likely terms:
$$w^* \sim \Pr(w_n \mid w_{n-k}, \dots, w_{n-1})\mathbf{1}(w_n \in \text{top-}k)$$
- *Temperature sampling*: sampling is limited to likely terms by raising the probabilities to a power:
$$w^* \sim \Pr(w_n \mid w_{n-k}, \dots, w_{n-1})^{\frac{1}{T}}$$
Here, T is the temperature (higher temperature means a more uniform sampling).
- *Beam search*: this technique searches forward one step at a time for the most likely sequence $(w_n, w_{n+1}, w_{n+2}, \dots)$ while limiting the search space to a maximum set of k candidate sequences.

Greedy techniques always produce the same text, while sampling generates different text each time. Output from lower-order n-gram language models may produce text that is non-sensical but potentially grammatical.

4.1.2 Language model evaluation

To determine if one language model is better than another, we use two main evaluation approaches:

- *Extrinsic*: the model is used in a downstream task, and the performance is evaluated based on the task's outcomes.
- *Intrinsic*: the model's parameters are trained on a dataset, and its performance is evaluated on a held-out dataset. The likelihood of the model producing the observed data is used to assess how well it is performing.

Definition (*Perplexity*)**.** Perplexity is a measure of how well a language model predicts new text.

It quantifies the level of surprise or confusion when encountering new data and reflects how unlikely the observed data is under the model. The perplexity is computed through the following steps:

1. Compute the probability of the observed sequence under the model.
2. Normalize the probability for the length of the text sequence.
3. Invert the probability to calculate uncertainty. Minimizing perplexity is equivalent to maximizing probability, so a lower perplexity indicates a better model.

Perplexity is closely related to other metrics used in training and evaluating predictive models:

- *Negative Log Likelihood* (nLL): the negative logarithm of the probability of the sequence. When divided by the sequence length, this gives the per-word nLL. Perplexity can be derived as:

$$\text{perplexity} = 2^{nLL}$$

- *Crossentropy*: the expected log surprise under the model. It represents the number of bits needed to quantify the surprise or uncertainty of a sequence.

4.1.3 N-gram limitations

As the value of n increases in n-gram models, the probability of encountering a specific sequence in the training corpus decreases exponentially. This results in sparse data, where many possible n-grams are never seen during training. When the model backs off to shorter n-grams, this significantly limits its ability to make accurate predictions.

Moreover, to generate reasonable and coherent language, we need to model long-distance dependencies, where the relationship between words may span across many tokens. Unfortunately, as n grows, both memory and data requirements scale exponentially with the length of these dependencies, making traditional Markov models impractical for capturing such long-range relationships.

4.2 Word embeddings

Word embeddings, which emerged around 2013, significantly improved performance on nearly every NLP task. They are dense vector representations of words in a high-dimensional space, typically with between 100 and 1000 dimensions. These embeddings are much more compact compared to the sparse one-hot encoding of terms, which typically requires a vector the size of the entire vocabulary. Given that document collections often have vocabularies ranging from 100,000 to 1 million tokens, word embeddings provide a much more efficient way to represent words.

Similar to one-hot encodings, word embeddings can also be aggregated to represent larger units of text. This aggregation allows for the capture of semantic meaning in a more computationally efficient manner.

4.2.1 Training

Word embeddings are produced using supervised ML models, specifically models that are trained to predict a missing word based on the surrounding context. The context can either include only previous words (causal models) or both previous and future words (non-causal models). In this setup, the training process can be described as follows:

- *Features*: the words in the current context.

- *Target*: the missing word that we aim to predict from the sequence.

This is essentially a multi-class classification problem, where the model needs to estimate the probability for every word in the vocabulary being the missing word.

Challenges A key challenge is that even a simple linear classifier for this task requires a large number of parameters. For example, if we use a multi-class linear classifier to predict the missing word, with a bag-of-words feature vector (ignoring word order), the model will require a parameter vector of the size of the vocabulary for each vocabulary term. Therefore, the total number of parameters will scale quadratically with the size of the vocabulary.

This quadratic growth in parameters was a significant issue before DL techniques emerged, as the vocabulary size in traditional NLP tasks was often very large. However, with modern DL techniques, we can overcome this limitation more efficiently.

4.2.2 Properties

Word embeddings possess several intriguing and sometimes surprising properties that make them powerful tools for NLP tasks:

- *Semantic clustering*: neighbors in the embedding space are often semantically related. Words that share similar meanings or appear in similar contexts are typically located near each other in the embedding space.

- *Dense and distributed representation*: word embeddings use multiple dimensions to capture semantic relationships. However, individual dimensions of the embedding vector are generally not directly interpretable.

- *Meaningful translations*: despite the individual dimensions being hard to interpret, translations in the embedding space have meaningful semantic implications.

- *Additive semantics*: certain semantic relationships can be represented as additive vectors.

- *Analogies*: word embeddings can encode analogies between words.

- *Discovering relationships*: embeddings can uncover various relationships between words, such as synonyms, antonyms, and other semantic connections, purely based on how words co-occur in text. These relationships are often encoded in the geometry of the embedding space.

The low-dimensional nature of word embeddings means that semantically similar terms tend to have similar representations in the vector space. This allows the model to generalize better from semantically related examples, improving its ability to handle unseen data. Moreover, embeddings place similar concepts close together in the vector space, making them useful for discovering implied (but unknown) properties of concepts.

4.2.3 Word2Vec

ord2Vec, developed in 2013 by Mikolov et al., is one of the most popular and influential word embedding models. It builds on early work by Bengio et al. in 2003 and was later followed by GloVe (2014) by Pennington et al. Word2Vec solved the problem of the large parameter space in traditional models by using a bag-of-words representation.

There are two main versions of Word2Vec:

- *Continuous Bag Of Words*: this version is trained to predict an observed word based on the surrounding context words. The context consists of all terms occurring within a symmetric window around the target word. The model predicts the target word c given the observed words w by applying a softmax function over the dot product of the word embeddings. Directly optimizing this model is computationally expensive because it requires summing over all possible target words. To address this, Mikolov et al. introduced negative sampling, where they sample some negative examples (words that were not observed) and turn the problem into a binary classification task. This greatly reduces the computational load. Although on modern GPUs, optimizing softmax directly is no longer a problem.

- *Skip-gram*: in this version, the model is trained to predict the observed words given a single context word. It works in exactly the same way as CBOW, but here the context is the sum of the word vectors around the target word.

In summary, Skip-gram is a 1-to-1 prediction model, while CBOW is a many-to-1 prediction model.

Word embeddings from Word2Vec can be viewed as a form of matrix decomposition. The model uses a square co-occurrence matrix, where each cell in the matrix represents the co-occurrence of a pair of words within a fixed-size context window. By factorizing this matrix, Word2Vec generalizes the information from these co-occurrence windows to produce the word embeddings.

GloVe GloVe (Global Vectors for Word Representation) is another model for generating word embeddings. It offers a probabilistic interpretation for the translation of words in the embedding space. The training objective of GloVe is formulated as fitting an objective function that is approximated by minimizing a weighted least squares objective, with several tricks required to ensure convergence. GloVe improves upon Word2Vec by focusing on global co-occurrence information, rather than just local context windows, making it a more globally-informed method for generating word embeddings.

Word2Vec and GloVe Word2Vec is typically faster to train, has lower memory requirements, and produces more accurate models, especially with the skip-gram approach. GloVe focuses more on capturing global word co-occurrences, making it different from the local context modeling of Word2Vec. According to Levy et al., skip-gram tends to outperform CBOW, except for when using FastText (a variant of Word2Vec).

4.2.4 FastText

Word embeddings work well when the vocabulary is fixed, but they face challenges when dealing with new or unseen words in the test set. If a word is not present in the trained model's vocabulary, we don't have an embedding for it and, traditionally, would have to ignore it. This is problematic, especially since we can often infer the meaning of the word from the letters or characters contained within it.

FastText, introduced by Bojanowski et al. in 2016, provides an elegant solution to this problem. Instead of learning embeddings for entire words directly, FastText splits words into smaller fixed-length character sequences (subword units), specifically character n-grams. This allows FastText to learn embeddings for these subword units, and then combine these embeddings to form the representation of the entire word. FastText, therefore, is a powerful extension of traditional word embeddings, as it can deal with the dynamic nature of language and is particularly well-suited to languages with rich morphology.

4.2.5 Usage

In causal models, the context words are restricted to those that occur before the missing word, which makes these models suitable for language modeling. In this setup, the model predicts the next word in a sequence based on the preceding words, allowing it to capture longer dependencies than traditional n-gram models. This capability enables a more sophisticated understanding of word sequences and improves the quality of text generation and prediction tasks.

In non-causal models, the context can include both previous and future words. These models are often used to generate word embeddings that can serve as additional feature vectors to represent words. The embeddings can significantly improve performance across a wide range of tasks, such as:

- Training classifiers for sentiment analysis or other classification tasks.
- Machine translation, where embeddings are used to translate text from one language to another by leveraging the semantic meaning of words.

By incorporating semantic knowledge from the context, these models benefit from the rich relationships between words, improving the quality of predictions and translations.

4.2.6 Vector database

Vector databases are specialized systems designed to index and retrieve objects based on their embeddings, enabling efficient similarity searches. These databases excel at performing fast nearest-neighbor searches in high-dimensional embedding spaces, which is essential for various applications.

However, finding nearest neighbors in high-dimensional spaces presents significant challenges. In such spaces, vectors tend to become approximately equidistant and orthogonal, making it difficult to distinguish close neighbors from distant ones. Traditional indexing methods, such as k-d trees, which provide $\mathcal{O}(\log(n))$ search complexity in low-dimensional spaces, degrade to $\mathcal{O}(n)$ behavior in high dimensions due to the curse of dimensionality.

To address these challenges, advanced algorithms have been developed that focus on approximate nearest neighbor (ANN) search techniques. One prominent approach is the use of Hierarchical Navigable Small World (HNSW) graphs. HNSW leverages a multi-layered graph structure to efficiently partition and navigate the space:

1. *Navigable Small World Graphs*: nodes in the graph are connected to their nearest neighbors, allowing for rapid traversal during searches.
2. *Hierarchical layers*: the graph consists of multiple layers, with all nodes present in the bottom layer and progressively fewer nodes in higher layers. This hierarchical structure enables efficient exploration by narrowing down candidate neighbors at each level.
3. *Search algorithm*: the algorithm begins at the topmost layer, identifies promising candidates, and iteratively refines the search by moving to lower layers until the nearest neighbor is found in the bottom layer.

4.3 Sequence labelling

Word order is crucial for understanding the meaning of text and for tasks like classification. While n-grams can help capture word order, they are inherently limited in length and may fail to fully represent complex dependencies.

Definition (*Sequence classification*)**.** Sequence classification takes an ordered sequence of tokens as input and produces a single prediction for the entire sequence.

Definition (*Sequence labelling*)**.** Sequence labeling takes an ordered sequence of tokens as input and generates a corresponding sequence of predictions.

Historically, sequence labeling has been tackled using:

- *Hidden Markov Models*: these function similarly to Naïve Bayes but applied to sequences. An Hidden Markov Model consists of: hidden states (unobserved), observed words (the actual text), transition probabilities (likelihood of moving between hidden states), and emission probabilities (likelihood of words appearing in specific states). Parameter estimation is typically done by counting frequencies in labeled data. If hidden states are unknown, the Expectation-Maximization algorithm can be used.
- *Conditional Random Fields*: these operate similarly to Logistic Regression but for sequential data. Instead of using transition and emission probabilities like Hidden Markov Models, Conditional Random Fields employ undirected potentials:
  - $\phi(t_1, t_2)$ for transitions between labels.
  - $\phi(t_1, w_1)$ for label-word relationships.

  By relaxing the generative assumption, Conditional Random Fields often achieve better performance while keeping parameter estimation similar.

Recent advancements leverage Recurrent Neural Networks to further improve sequence labeling performance by capturing long-range dependencies more effectively.

4.3.1 Recurrent Neural Networks

Word embeddings represent words in a continuous semantic space. To aggregate embeddings and represent an entire document, one approach is to sum them, similar to how one-hot encodings create a bag-of-words representation. However, this method ignores word order, leading to documents with different structures but similar words having the same representation.

Recurrent Neural Networks provide a more effective way to accumulate information across a document while preserving word order. They achieve this by sequentially combining the embedding of the current word with the context from previous words.

Recurrent Neural Networks operate using a simple yet powerful structure. They take two vectors as input: the current input and the previous state. They then produce two vectors as output: the current output and the updated state. This structure allows Recurrent Neural Networks to process arbitrarily long input sequences and encode them into a single embedding.

4.3.2 Long Short-Term Memory

Long Short-Term Memory (LSTM) is an advanced variant of RNNs designed to learn context and capture long-range dependencies. It achieves this through a gating mechanism that controls the flow of information:

- Information passes through by default unless explicitly modified.
- New information can be added to the state.
- Irrelevant information can be removed (forgotten).

LSTMs learn when to remember, forget, and output information at each time step, making them highly effective for sequential data.

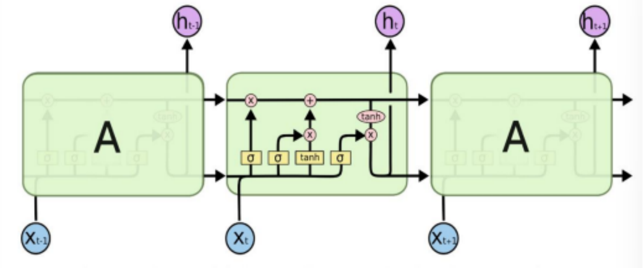


Figure 4.1: Long Short-Term Memory

LSTMs can be stacked to form deeper networks, enhancing their ability to handle nested contexts. This capability is particularly useful for processing natural language, where understanding hierarchical structures and long-term dependencies is essential.

4.3.3 Applications

Sequence classifiers and sequence labelers are widely used in various NLP tasks.

Part of Speech tagging Modern grammar categorizes words into open and closed classes.

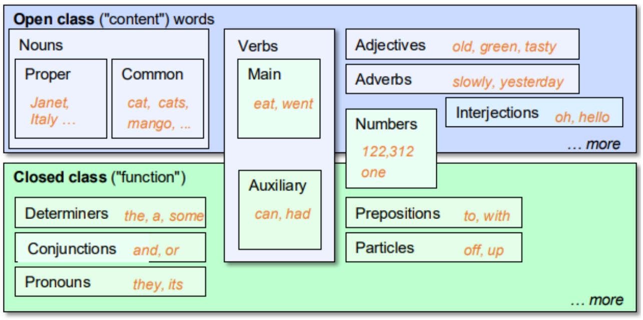


Figure 4.2: Modern grammar

POS tagging involves assigning a part-of-speech label to each token in a sequence. This is useful for:

- Developing features for NLP models.

- Reducing ambiguity in bag-of-words representations by appending POS tags to word occurrences.
- Serving as a foundational step for other NLP tasks, such as syntactic parsing and linguistic analysis.
- Supporting applications like text-to-speech and the study of linguistic chang.

POS tagging maps a sequence of words $(x_1, \ldots, x_n)$ to a sequence of POS tags $(y_1, \ldots, y_n)$ Although 85% of English vocabulary terms are unambiguous, about 60% of tokens in actual text are ambiguous, making POS tagging a challenging task. However, modern systems achieve an accuracy of around 97%, which is comparable to human performance.

Named Entity Recognition Named Entity Recognition is the task of identifying entities mentioned in a text. It is typically framed as a sequence labeling problem and serves as a crucial step in extracting structured knowledge from unstructured text.

A named entity refers to an object in the real world. Common entity categories include: PER (person), LOC (location), ORG (organization), GPE (geo-political entity). Entities are often multi-word phrases, and the term named entity has been extended to cover concepts beyond just real-world objects.

Named Entity Recognition involves two key steps identify spans of text that represent proper names and assign a label that categorizes the type of entity.

Traditional Applications of Named Entity Recognition includes:

- *Sentiment analysis*: detecting sentiment toward something.
- *Information extraction*: extracting structured facts about entities from raw text.
- *Question answering*: understanding entity-related questions and retrieving relevant information.
- *De-identification*: removing personal references from text to ensure privacy.

The main challenges in Named Entity Recognition are:

1. *Segmentation*: unlike POS tagging, where each word has a single tag, named entities can span multiple words.
2. *Type ambiguity*: the same word or phrase can have different meanings depending on context.

To transform NER into a sequence labeling task (assigning one label per token), the Begin-Inside-Outside (BIO) tagging scheme is commonly used:

- *Begin* (B): the first token in an entity span.
- *Inside* (I): tokens inside the entity span.
- *Outside* (O): tokens that do not belong to any entity.

This approach enables models to correctly identify and classify multi-word entities within a text.

Entity Linkage Identifying a named entity in text is only the first step. The next challenge is determining which real-world entity the mention refers to, a process known as entity linkage. This task is difficult due to ambiguity.

Entity linkage methods rely on the relative importance of entities and context within the text, including other mentioned entities that provide clues.

Entity linkage typically uses structured knowledge sources. However, many individuals or objects lack a source. In such cases, custom ontologies are used for better accuracy.

Relation extraction Once entity mentions are correctly linked to real-world entities, the next step is relation extraction—identifying relationships between entities to build structured knowledge. Extracted relationships can be used to populate a knowledge graph or knowledge base. This is often framed as a problem of predicting missing links in a graph. Entity embeddings help model relationships, as spatial transformations in embedding space naturally encode relational patterns.

4.3.4 Parse trees

Parse trees (also called syntax parse trees or dependency parse trees) represent the structure of a sentence based on a formal grammar. These grammars define a set of rules for generating valid text and are commonly used for analyzing both natural language and programming languages. Given a piece of text, parsing reverses the generative process by identifying which grammatical rules were applied and the order in which they were applied This recursive process results in a tree structure for each sentence, where each node represents a syntactic component.

Parse trees help determine how words in a sentence relate to one another. This structural analysis allows us to infer the intended meaning (semantics) of the sentence.

In theory, formal grammars alone could be used to parse text. However, natural language is inherently ambiguous, and formal grammars tend to be brittle (struggling with variations in phrasing). In practice, ML techniques are often needed to extract accurate parse trees from real-world text.

Understanding sentence structure is crucial for many NLP tasks, including: populating structured databases, generating coherent text, and extracting relationships

4.3.5 Co-reference, taxonomy and ontology

Definition (*Co-reference*)**.** Co-reference resolution is the task of determining who or what a given pronoun or noun phrase refers to within or across sentences.

In most cases, a pronoun appears after its referent in the text. However, there are instances where the pronoun appears before the referent, requiring more complex resolution strategies.

It helps in understanding what is being said about an entity, especially when pronouns are used. It is crucial for tasks such as information extraction, chat bots, and text summarization, where accurate entity tracking is needed.

Definition (*Taxonomy*)**.** Taxonomy is the hierarchical structure of concepts.

Definition (*Ontology*)**.** Ontology is the formal representation of concepts and their relationships.

Ontologies typically consist of the following components: classes, individual, attributes, relationships, and logical rules.

In an ontology or knowledge base, the relationships between concepts form a graph structure. These knowledge representations capture the factual information conveyed in sentences, enabling better comprehension and reasoning.

Ontologies and knowledge bases typically follow open world semantics, where any statement not known to be true is simply unknown. This contrasts with the closed world assumption used in databases, where any statement not known to be true is assumed false.

4.4 Sequence-to-sequence models

RNNs and their more advanced variant, LSTMs, proved to be so powerful that they quickly became the go-to solution for sequence-to-sequence (seq2seq) tasks—problems where the input is a sequence of data and the output is another sequence. One of the most prominent applications of seq2seq models is machine translation, where an input sentence in one language is translated into another language. To build a translation model using LSTMs, two distinct RNN-based components are typically trained:

- *Encoder*: this component processes the input sequence and generates a compact representation of the entire sequence. Essentially, it encodes the meaning of the input into a fixed-size vector.

- *Decoder*: takes this encoded representation as its starting point and generates the output sequence word by word. It essentially decodes the meaning captured by the encoder into the target language or desired output format.

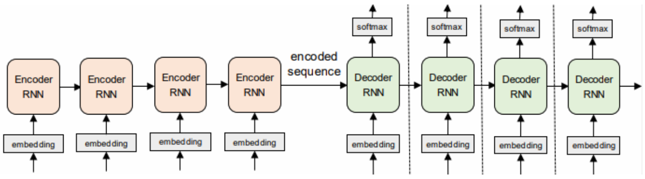


Figure 4.3: Sequence to sequence

This encoder-decoder framework revolutionized the field of NLP and set new state-of-the-art performance benchmarks across a wide range of tasks. In fact, many NLP problems can be framed as seq2seq tasks.

One major issue is that the encoder must compress all the information from the input sequence into a single fixed-size vector, which is then passed to the decoder. This bottleneck can lead to information loss, especially when dealing with long sequences. As a result, the decoder might struggle to generate accurate translations or outputs because it doesn't have direct access to the full context of the input sequence

4.4.1 Attention

Attention mechanisms have become a cornerstone of modern approaches to both text and image processing. In computer vision, attention allows models to focus on specific regions of an image when making predictions. Similarly, in NLP, attention enables models to concentrate on specific parts of the input text that are most relevant for generating each part of the output.

The key idea behind attention is to make the encoded input available to the decoder in a way that provides a direct route for information to flow from the input to the output. This addresses a critical limitation of traditional encoder-decoder architectures: they rely on compressing the entire input sequence into a single fixed-size vector, which can lead to information loss, especially for long sequences.

Mapping problems Directly mapping input words to output words is problematic for several reasons:

- *Variable token lengths*: different languages often require different numbers of tokens to express the same concept.
- *Word order differences*: languages frequently use different word orders, making one-to-one mappings impractical.
- *Contextual dependencies*: generating the correct output word often requires knowledge not just of the current input word but also of future words in the sentence.

Attention solves these challenges by providing a mechanism to pass information from the embeddings of input words to corresponding output words dynamically. The flow of information into the decoder is controlled by the previous state of the decoder itself.

Computation Attention computes a similarity score between the decoder's current state and the embeddings of each input word. These scores determine how much attention should be paid to each input word when generating the next output word. Mathematically, this process can be described as follows:

1. *Similarity computation*:

   $$w_{ij} = \Pr(j \mid i) = \mathrm{softmax}(\mathrm{similarity}(\mathbf{h}_{i-1}, \mathbf{e}_j))$$

   Here, w_{ij} represents the weight assigned to the j-th input word when generating the i-th output word.

2. *Weight average*: soft attention computes a weighted average over the input embeddings:

   $$\mathbf{z}_i = \sum_j w_{ij} \mathbf{e}_j$$

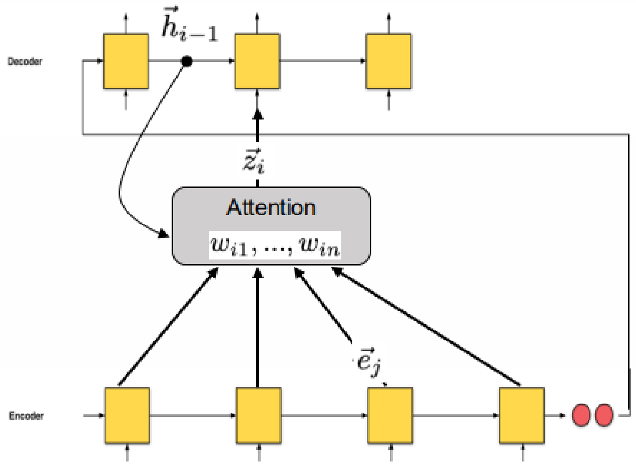


Figure 4.4: Attention

Similarity There are two common methods for computing similarity between the decoder state and input embeddings:

- *Additive attention*: similarity is computed using a feed-forward neural network:

$$\text{similarity}(\mathbf{h}_{i-1}, \mathbf{e}_j) = \text{FFNN}(\mathbf{h}_{i-1}, \mathbf{e}_j)$$

- *Multiplicative attention*: similarity is computed as the dot product between the decoder state and the input embedding, normalized by the square root of the embedding dimension d to ensure stable gradients:

$$\text{similarity}(\mathbf{h}_{i-1}, \mathbf{e}_j) = \frac{\mathbf{h}_{i-1} \cdot \mathbf{e}_j}{\sqrt{d}}$$

Once the similarity weights are calculated, they are used to compute the weighted sum of the input embeddings:

$$\mathbf{z}_i = \sum_j \text{softmax}\left(\frac{\mathbf{h}_{i-1} \cdot \mathbf{e}_j}{\sqrt{d}}\right) \mathbf{e}_j$$

Query-key-value Attention can be generalized using the query-key-value framework:

$$\mathbf{z}_i = \sum_j w_{ij} \mathbf{v}_j = \sum_j \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}}\right) \mathbf{v}_j$$

Here:

- *Query* ($\mathbf{q}_i$): represents what the model is looking for at position i.
- *Key* ($\mathbf{k}_j$): acts as an index to locate relevant information.
- *Value* ($\mathbf{v}_j$): contains the actual information stored at position j.

These components are typically transformed using learned linear projections:

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{h}_{i-1} \qquad \mathrm{k}_j = \mathbf{W}_k \mathbf{e}_j \qquad \mathbf{v}_j = \mathbf{W}_v \mathbf{e}_j$$

Here, $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v \in \mathbb{R}^{n \times n}$

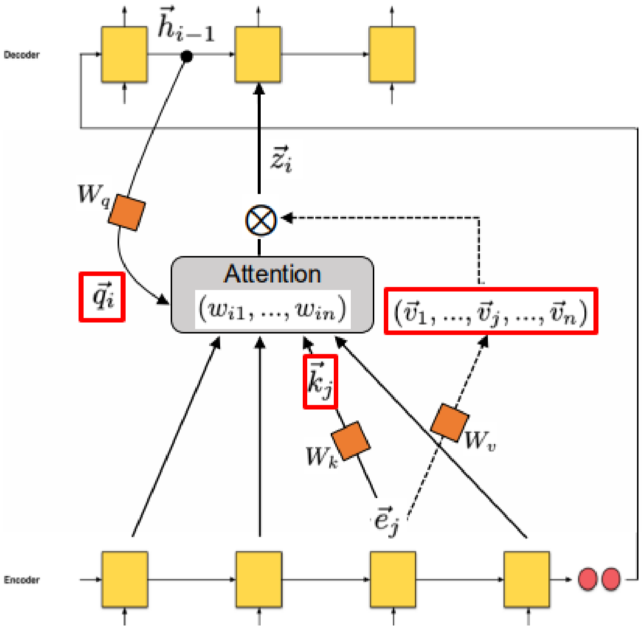


Figure 4.5: Query-key-value attention

4.4.2 Self-attention

Self-attention is a powerful mechanism that allows models to capture relationships between different parts of the input sequence without relying on recurrence or convolution. Deeper models generally outperform shallow ones because each layer builds on simpler features extracted by the layers below. However, training RNNs poses significant challenges:

1. Information must propagate from the first position of the encoder to the last position of the decoder, and gradient information must flow back along the entire sequence during backpropagation.
2. RNNs process sequences sequentially, making parallelization difficult and resulting in training times that scale linearly with the length of the input ($\mathcal{O}(n)$).
3. Training deeper networks with many layers becomes increasingly difficult due to vanishing or exploding gradients.

Self-attention addresses these issues by removing the recurrent connections from the encoder and decoder. Instead, it relies on attention mechanisms to directly pass information between positions in the sequence.

Problems Removing recurrence introduces two main challenges:

1. *Query choice*: the current output of the encoder is used as the query instead of relying on the decoder's context.
2. *Word order loss*: positional encoding is added to the input embeddings to explicitly encode the order of words.

Computation Self-attention updates a sequence of embedding vectors based on the weighted average of incoming embedding vectors. At each position i, the mechanism computes:

- *Query*: a linear transformation of the embedding at position i.
- *Key*: a linear transformation of the embedding at position j.
- *Value*: a linear transformation of the embedding at position j.

The output embedding at position i is then computed as:

$$\mathbf{z}_i = \sum_j w_{ij}\mathbf{v}_j = \sum_j \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}}\right)\mathbf{v}_j$$

Applications Self-attention models are trained to perform tasks such as:

- *Masked Language Modeling*: recover missing words from the input text based on surrounding context. Input text is corrupted by randomly masking certain tokens, and the model learns to predict them.
- *Next word prediction*: predict the next word in a sequence based on the previous words.

4.5 Transformer

The original Transformer model, introduced in the seminal 2017 paper Attention Is All You Need, marked a revolutionary shift in the field of NLP.

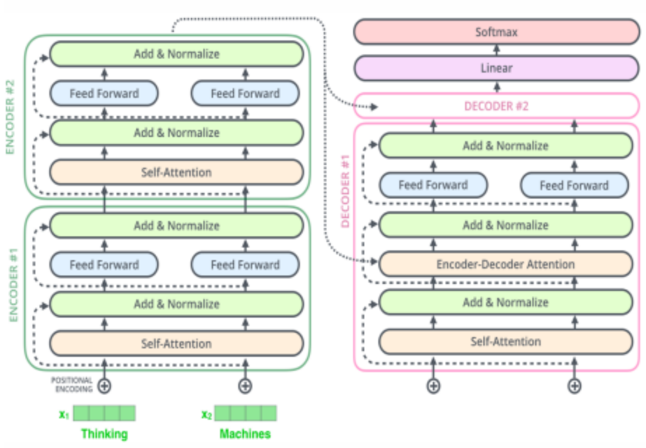


Figure 4.6: Transformer

At its core, the Transformer relies on a self-attention mechanism as its primary building block. This mechanism allows the model to dynamically focus on different parts of the input sequence when processing each token. The basic self-attention module consists of the following components:

1. *Multiple attention heads*: instead of relying on a single attention mechanism, the Transformer employs multiple attention heads that operate in parallel. Each head computes attention independently, capturing different types of relationships between tokens. These outputs are then concatenated and linearly transformed to produce the final result.

2. *Feed Forward Neural Network*: after the multi-head attention step, the output is passed through a Feed Forward neural network. This network applies a non-linear transformation to the data, allowing the model to learn more complex patterns.

3. *Residual connections and layer normalization*: to facilitate training and improve gradient flow, the Transformer uses residual connections and layer normalization after each sub-layer.

4. *Positional encoding*: this encoding provides information about the position of each token in the sequence, enabling the model to understand the sequential nature of language. Simplest way to do that would be to use a binary encoding of the position Since the embedding vector is made of floating point values, makes more sense to encode positions using sinusoids.

The basic self-attention module is stacked multiple times to form the full Transformer architecture. This stacking allows the semantics of each token to build up progressively over multiple layers. Each layer refines the representation of the input tokens by incorporating information from other parts of the sequence, resulting in rich, context-aware embeddings.

Self-attention Self-attention is so valuable for language models. Words often have multiple meanings, and their interpretation depends heavily on the surrounding context. Self-attention solves this problem by allowing a word's representation to adapt based on its context. It does this by learning a weighting function that prioritizes the most relevant parts of the input sequence when constructing a word's meaning. Essentially, it enables the model to focus on different words or phrases in the sentence, assigning more importance to those that matter most for understanding the current word. This mechanism becomes even more powerful when stacked across multiple layers. With each layer, the model refines its understanding of relationships between words, enabling it to tackle complex linguistic tasks like co-reference resolution.

4.5.1 Architecture

The Transformer architecture is composed of three main components:

- *Input module*: this module generates the initial embedding for each token in the input sequence.

- *Transformer blocks*: multiple transformer blocks are stacked on top of each other. Each block refines the embeddings by incorporating information from the surrounding context. Specifically, a transformer block modifies the embeddings through two key subcomponents:

    1. *Self-attention block*: captures relationships between tokens by allowing each token to attend to others in the sequence. Contains multiple self-attention heads, each operating independently on a reduced embedding size of $\frac{d}{h}$, where d is the original embedding dimension and h is the number of attention heads. The self-attention mechanism relies on three key components:

        – *Query* (Q): represents the current token whose representation we are updating.
        – *Key* (K): encodes other tokens in the sequence, helping determine their relevance to the query.
        – *Value* (V): provides the actual content used to update the query's representation.

        These components (Q, K, V) are produced by applying linear transformations (matrices) to the original embeddings:

        $$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

        Here, $d_k = \frac{d}{\text{number of parallel attention heads}}$

    2. *Feed-Forward Neural Network*: a simple, position-wise neural network that further processes the updated embeddings. Typically has a fan-out factor of 4, meaning the hidden layer is four times the size of the input embedding.

- *Output module*: after the embeddings have been refined through multiple transformer blocks, this module decodes them to predict the next word or perform other downstream tasks.

4.5.2 Transformer input

Choosing between word-level and character-level representations involves balancing expressivity, sequence length, and the model's ability to generalize. We have the following possibilities:

- *Word-level tokens*: semantically rich and result in shorter sequences, reducing computational complexity. However, they lead to larger vocabularies, struggle with out-of-vocabulary words, and ignore morphological details like prefixes or suffixes.

- *Character-level tokens*: more flexible and capable of handling any word form, produce much longer sequences, increasing inference time. They also place the burden on the model to learn word structures from scratch, often leading to less interpretable embeddings.

- *Sub-word tokens*: uses data-driven methods like Byte-Pair Encoding to identify frequent character sequences. This approach captures common prefixes, suffixes, and word fragments, balancing vocabulary size and the model's ability to handle diverse linguistic patterns effectively.

4.5.3 Bidirectional Encoder Representations from Transformers

Bidirectional Encoder Representations from Transformers (BERT) revolutionizes text representation by learning deep contextualized embeddings through a masked language modeling approach. During pre-training, BERT randomly masks words in the input using a special token and trains the model to predict these masked words. This process helps BERT understand the relationships between words in a sentence, capturing rich linguistic patterns. Pre-trained on large corpora like Wikipedia and books, BERT provides powerful text representations without requiring manual feature engineering The advantages of this model are:

- *Eliminates feature engineering*: unlike traditional methods that rely on handcrafted features, BERT automatically learns meaningful representations.

- *Preserves word order*: by leveraging bidirectional context, BERT retains word order and contextual relationships, outperforming count-based approaches.

- *Unsupervised pre-training*: BERT benefits from unSupervised Learning on vast amounts of text, enabling it to generalize well even with limited task-specific data.

Fine tuning During pre-training, a special token is added at the start of every input sequence, but it is unused in the loss function. However, during fine-tuning, this token becomes crucial because the model is trained to produce a class label in place of the token. Fine tuning works because BERT is pre-trained on massive datasets, it requires small data and has multilingual support.

BERT can be used for text classification, sequence labelling and text pair classification and even question-answering tasks.

4.5.3.1 Document similarity

BERT-based models can be effectively utilized to re-rank documents in web search tasks by predicting the relevance of documents to a given query. This process involves fine-tuning the BERT model on a dataset of $\langle \text{query}, \text{document} \rangle$ pairs, where the goal is to predict a relevance label for each pair. To structure the input, the [SEP] token is used to separate the query and document, enabling the model to distinguish between the two components.

In addition to relevance prediction, BERT can also be fine-tuned to estimate semantic similarity between documents. This can be achieved through the following steps:

- Start with a dataset consisting of pairs of similar documents and randomly selected pairs of (likely) dissimilar documents.

- Fine-tune a pairwise classifier to distinguish between similar and dissimilar document pairs.

- Use the logits or the predicted probability of the ”similar” class as a measure of similarity between documents.

If ground truth similarity scores or distance metrics are available, the model can alternatively be fine-tuned for a regression task. In this case, the objective is to directly predict the similarity score, providing a more precise measure of document similarity.

4.5.3.2 Sentence transformer

Using a pairwise BERT classifier to estimate the relevance of documents for a given query is highly effective due to BERT's ability to leverage the word order in both the query and the document and capture the semantics of words using contextual embeddings. However, this approach is computationally expensive because BERT performs numerous matrix multiplications during inference. It requires GPU acceleration to achieve reasonable speeds but still incurs significant latency. A relevance score must be computed for every document in the collection, which becomes infeasible at scale.

To address these challenges, we aim to speed up computation by performing as much precomputation as possible. However, since similarity scores depend on the query, precomputing them beforehand is not feasible. Below are two practical solutions:

1. *Lexical search with BERT re-ranking*: use a fast lexical search engine to quickly retrieve a candidate set of relevant documents. Apply the fine-tuned pairwise BERT classifier only to re-rank this smaller subset of candidates, significantly reducing computational overhead.

2. *Precompute document embeddings*: train BERT to generate embeddings for entire documents, compute similarity between documents using the dot product of their embeddings. Then, train the model on pairs of similar and dissimilar documents using a contrastive loss function, which gives high similarity scores for similar document pairs. Due to BERT's context length limitation , it may be necessary to split long documents into sections, compute embeddings for each section, and aggregate them.

Sentence-BERT (SBERT) is a specialized framework that addresses these challenges by training BERT to learn fixed-length vector representations for entire sentences or documents. It uses contrastive learning to create an embedding space where semantically similar documents produce similar embeddings. In practice, SBERT trains two BERT models—one for encoding queries and another for encoding documents—and computes the similarity between them using the dot product (or cosine similarity) of their [CLS] token embeddings. The model is trained on pairs of similar and dissimilar documents to ensure that similar documents yield high similarity scores and dissimilar documents yield low similarity scores.

4.5.4 Generative Pre-trained Transformer

Generative Pre-trained Transformer (GPT) is an auto regressive language model designed to predict the next token in a sequence. It achieves this by masking future tokens during training, ensuring the model only uses past and present context to make predictions. This approach makes GPT particularly effective for text generation , as it learns to produce coherent and contextually relevant sequences by predicting one word at a time. Unlike BERT, which uses bidirectional context, GPT predicts the next word based solely on preceding tokens, making it ideal for generating fluent and natural-sounding text.

Fine-tuning While GPT-2 can serve as a text encoder for classification tasks, its true strength lies in text generation. Therefore, it is particularly well-suited for tasks such as translation, summarization, dialogue generation, and more. During the fine-tuning process, specific strategies are employed to adapt the model effectively for these tasks.

4.5.4.1 Few shot learning

Language models are highly versatile universal learners that can be utilized effectively with or without fine-tuning. During pre-training, these models are exposed to a vast number of examples of few-shot learning scenarios, enabling them to generalize and adapt to various tasks with minimal additional guidance. The ability to predict text serves as a flexible foundation for delivering a wide range of functionalities:

- *Translation*: even without explicit training for machine translation, models like GPT-2 have demonstrated the ability to perform translation tasks reasonably well. For instance, despite being trained primarily on an English corpus, GPT-2 could translate between languages because some multilingual data was inadvertently included in its training set. While not the best-performing translation system, this showcases the model's inherent adaptability.

- *Question Answering*: language models can store factual knowledge within their parameters and use it to answer questions. For example, GPT-2 often provided accurate answers to questions, even though it was not explicitly trained for this task. However, while confident predictions were usually correct, the reliability of such responses was not on par with dedicated question-answering systems at the time.

- *Reading Comprehension*: by providing source context—such as a document containing relevant information—language models can extract answers and demonstrate reading comprehension. This capability extends to tasks like fact-checking, where the model retrieves potential evidence supporting or refuting a claim by using the provided context.

- *Summarization*: language models can generate concise summaries of longer texts, leveraging their understanding of context and relevance to distill key information effectively.

4.6 Multi modal models

Aligning the embedding spaces is crucial for multi-modal models. Embeddings can be generated not only for text. By employing contrastive learning, we can force these two distinct embedding spaces to align with each other. The process involves taking a collection of image-text pairs, and training a classifier on batches of these pairs. Specifically, the classifier learns to identify which piece of text corresponds to which image and vice versa. Aligning the embedding space in this manner enables powerful applications like semantic image search using text queries.

4.6.1 Multi-task learning

Language models are highly versatile and have been adapted for multi-task learning. Research has shown that models trained on multiple tasks often outperform those specialized for a single task. Some approaches even aim to learn the optimal prompt for each specific task. In the context of multi-modal learning, the transformer architecture demonstrates remarkable flexibility. It is relatively straightforward to extend text-to-text models to handle multi-modal settings, where both text and images are processed together. This capability facilitates the learning of tasks across different types of media, enabling more comprehensive and integrated models.

4.7 Large Language Models

Language Models (LMs) are models designed to predict the next token in a sequence. As the name suggests, Large Language Models (LLMs) are simply very large versions of these models. After the Transformer architecture demonstrated its power for language modeling tasks, a race began to build increasingly larger models. Although the ultimate limits of scaling are not fully understood, empirical results show that the performance of GPT-style architectures improves approximately logarithmically with both the amount of training data and the total training time.

4.7.1 Chatbot

While large language models (LLMs) and chatbots are closely related, they are not the same. LLMs are primarily trained to predict the next token in a sequence of text, while chatbots are fine-tuned specifically for conversational interaction with users.

One major technique for turning an LLM into a chatbot is Reinforcement Learning from Human Feedback (RLHF). In this process, the model is exposed to many conversations with real users, receiving feedback on which responses were most appropriate, helpful, or satisfying. Over time, the model adjusts its behavior, learning to generate answers that better align with human preferences. Rather than simply continuing text, chatbots are designed to produce responses that are more engaging, polite, and useful.

Another important concept is instruction tuning, where a model is trained to perform a variety of tasks by following natural language instructions. Instead of optimizing only for token prediction, the model learns to interpret prompts framed as explicit commands or questions. This approach makes the model more versatile and better able to generalize across a wide range of user requests.

4.7.2 Prompting

During fine-tuning, language models are trained to engage in structured conversations with users. This involves recognizing special tokens that delineate different parts of the dialogue. Conversations are typically made up of three types of messages:

- System messages, which define how the chatbot should behave.

- User messages, which contain the user's input or requests
- Assistant messages, which represent the chatbot's responses.

Although LLMs fundamentally operate as text-in/text-out systems, all parts of the conversation—past and present—must be serialized into a single text sequence. This is done by concatenating messages with the appropriate formatting, often using chat templates that define the structure of the conversation at the token level.

A key component in this setup is the system prompt, which sets the rules for how the chatbot should respond. It guides the model on what behaviors to adopt and which topics or tones to avoid. This prompt plays a crucial role in ensuring safe and aligned responses, helping prevent the model from producing offensive or harmful content. System prompts are often proprietary and not publicly disclosed, though users have had varying success inferring them by querying models directly.

4.7.2.1 Chain-of-Thought reasoning

Introduced in 2022, chain-of-thought (CoT) prompting helps models perform better on tasks that require step-by-step reasoning. By encouraging the model to explain its reasoning as it answers, CoT prompting often leads to more accurate and interpretable results.

Even without providing examples, a simple phrase can trigger this behavior (an approach known as zero-shot CoT prompting). Taking it further, users have found that slightly extending this prompt or adding analogies can yield even better performance.

To improve answer quality, some methods involve generating multiple responses and selecting the most common one. Alternatively, the model can critique its own responses, regenerating them if it deems its previous answer incorrect. This form of self-reflection enhances both performance and trustworthiness.

4.7.2.2 Test-Time compute scaling

Recent models, such as Deepseek-R1, take a more structured approach to reasoning by separating the thinking and answering phases. Their outputs are formatted with dedicated tags, allowing the model to explicitly allocate time to reasoning before producing a final response. This technique, known as test-time compute scaling, encourages the model to spend more computational effort.

The training process involves exposing the model to difficult questions paired with answers, but without showing the steps needed to solve them. Over time, the model learns to generate increasingly detailed reasoning as part of its response, effectively training itself to think when necessary.

In some setups, a second model (possibly larger or more specialized) can be used to verify the final answer or evaluate the quality of the reasoning. This additional layer of oversight helps ensure that responses are not only accurate but also well justified—especially useful when ground-truth answers aren't available.

4.7.3 Limitations

The major limitations of LLMs are:

- *Hallucinations*: LLMs are known to sometimes produce fabricated or inaccurate information. These models are primarily optimized to generate content that is engaging and human-like, which can sometimes conflict with the objective of delivering strictly factual and truthful information. Hallucinations can manifest in many forms, including content that is not faithful to known facts, incoherent with provided data, or not logically derivable from available information.

- *Limited reasoning*: earlier versions of LLMs exhibited significant reasoning limitations. These issues sometimes arose from tokenization errors, or from inherent limitations related to the model's architecture. As a result, the models could struggle with multi-step logical reasoning or fail to maintain consistency across complex tasks.

- *Lack of robustness*: prompt sensitivity remains an active area of research. Small variations in prompt wording can lead to disproportionately large differences in model performance, making the behavior of LLMs sometimes unpredictable and frustrating for users. Generally, the clearer, more structured, and less ambiguous the prompt, the more reliable the model's output. However, ensuring consistent performance across a wide range of tasks is still a major challenge.

- *Jailbreak*: some users attempt to bypass the chatbot's safeguards. Ethics researchers may do this to test whether the model can be prompted to generate harmful or inappropriate content despite its safety constraints. Security researchers, on the other hand, may try to elicit memorized training data, which poses a significant privacy and security risk. This risk is exacerbated when user-provided content is incorporated into model fine-tuning.

4.7.4 Scaling laws

Numerous studies have explored how the performance of large language models scales with various factors. A near-linear relationship has been observed between model performance and the logarithm of: computation time, training dataset size, and number of model parameters. One key insight comes from the Chinchilla scaling law, which provides guidance for training Transformer-based language models efficiently. It suggests that, given a fixed computational budget (measured in FLOPs), optimal performance is achieved when the number of model parameters (N) and the number of training tokens (D) are scaled in roughly equal proportions.

Additionally, when training large models, it is crucial to adjust the learning rate appropriately. As model size increases, the learning rate should typically decrease proportionally. Various techniques and heuristics exist to estimate an optimal learning rate schedule based on model size and training dynamics.

4.7.5 Efficiency improvements

Recent research has introduced several architectural and algorithmic enhancements to improve the efficiency and scalability of Transformer-based models:

- *Layer Input Normalization*: normalization is applied to the input of each sub-layer (self-attention and feed-forward blocks), rather than to the residual stream, which can improve training stability and model performance.

- *Positional Embeddings*: improvements form Absolute Positional Embeddings to Rotational Positional Embeddings:

  - *Absolute Positional Embeddings*: the original Transformer used sinusoidal position embeddings added to token embeddings. Learned absolute embeddings were also explored but showed limited gains. Absolute encodings fix the maximum sequence length, reducing generalization to longer contexts.
  - *Relative Positional Embeddings*: used in models like T5, a learned bias is added to the query-key similarity score, enabling better generalization to varying context lengths. However, this approach can slow down self-attention and complicate caching mechanisms.
  - *Rotational Positional Embeddings*: RoPE encodes position by rotating token embeddings. This method combines the computational simplicity of absolute encodings with the benefits of relative positioning, as vector dot products become dependent only on relative positions.

- *Grouped Self-Attention*: a memory-efficient variant where queries are shared across subsets of attention heads, reducing the number of parameters and computational cost.

- *Enhanced MLP Layers*: new architectures add an additional up-projection matrix with a linear activation, creating high-dimensional hidden representations (typically $\sim 4d$) that are summed before projection back to the original dimensionality. This more complex feed-forward structure improves model expressiveness.

- *Mixture of Experts*: techniques like the Switch Transformer use a routing mechanism to select a subset of expert FFNNs for each token. This allows the model to scale to billions of parameters without a corresponding increase in computation per token, thus improving performance while maintaining efficiency.

- *Sliding Window Attention for Longer Contexts*: recent methods use a sliding window approach to attention, enabling models to process much longer contexts. This allows for full-document understanding and greatly expands application domains.

For the efficient hardware deployment we have the following techniques:

- *Low-Bit quantization*: transformers typically require GPUs for fast inference, but GPU memory is limited. Model weights are commonly quantized to lower precisions to save memory and improve inference speed. While quantization reduces accuracy slightly, it enables deployment of larger models on constrained hardware.

- *Low-Rank Adaptation*: fine-tuning large models is memory-intensive and often infeasible on standard hardware. LoRA addresses this by learning low-rank updates to the weight matrices. A weight matrix W is adjusted via a low-rank decomposition: $W' = W + AB$, where A and B are much smaller matrices (with B initially zero). This drastically reduces the number of trainable parameters and memory required for fine-tuning, making it practical for smaller datasets and limited compute environments.

4.7.6 Integration

Retrieval-Augmented Generation (RAG) Traditional LLM-based chatbots are limited by the information encoded within their model parameters during training. This creates challenges when responding to: domain-specific queries requiring expert knowledge, and questions about recent events or updates unavailable at training time. RAG models address this limitation by augmenting LLMs with real-time retrieval capabilities. They can search external knowledge sources and then generate responses grounded in the retrieved content. This allows LLMs to remain up-to-date and provide more accurate, context-specific answers.

Agentic AI Agentic AI builds on the LLM's ability to reason about tasks and goals, enabling models to make decisions and take actions in pursuit of user-defined objectives. These systems plan and execute actions to achieve a goal and can modify external systems—for instance, updating a user's address in a database based on a natural language request. Frameworks such as `LangChain` facilitate the development of such agent-based applications by integrating LLMs with tools, memory, and actions in a programmable environment.

CHAPTER 5

Speech detection and generation

5.1 Introduction

Human speech consists of two primary categories of sounds:

- *Vowels*: produced without significant constriction in the vocal tract.
- *Consonants*: created by partially or fully closing parts of the vocal tract.

The distinct sound units that compose words are referred to as phonemes.

Source filter model A widely used model for human phonation is the source-filter model, which separates speech production into:

- A source component: the glottis generates a pulse train or noise-like excitation.
- A filter component: the vocal tract shapes this excitation to produce different sounds.

While the source contributes to voice characteristics, the filter carries most of the linguistic information and is therefore more critical for speech recognition tasks.

5.1.1 Time series representation

Speech is fundamentally a time series of air pressure variations. Different types of speech sounds have distinct time-domain characteristics:

- *Vowels*: periodic signals.
- *Fricatives*: consonants formed by forcing air through a narrow channel.
- *Glides*: smooth transitions between sounds.
- *Bursts*: sudden, rapid transitions.

To analyze these signals effectively, it's useful to transform them into the frequency domain using the Fourier Transform.

5.1.2 Spectrogram

To capture how the content evolves over time, we compute a spectrogram, which shows the frequency spectrum of short segments of the audio.

The Short-Time Fourier Transform (STFT) analyzes local frequency content by:

- Dividing the signal into overlapping chunks.
- Applying a window function to each chunk to reduce spectral leakage.
- Computing the Fourier transform of each windowed segment.

This results in a time-frequency representation of the audio signal.

Pre-Emphasis Filter Before performing the STFT, a pre-emphasis filter is often applied to amplify high-frequency components. This helps balance the spectrum, improve the signal-to-noise ratio (SNR), and mitigate numerical instability in the Fourier transform. A common implementation uses a first-order filter:

$$y[n] = x[n] - \alpha x[n-1]$$

Here, α is typically around 0.95.

5.1.3 Mel spectrogram

Human perception of pitch is non-linear: we distinguish frequencies based on their relative rather than absolute differences. To better match human hearing, we use the Mel scale, which maps frequency f in Hz to Mel units using:

$$\text{Mel}(f) = 2595 \log_{10}(1 + \frac{f}{700})$$

The Mel spectrogram enhances the traditional spectrogram by limiting the frequency range, mapping linear frequencies to the Mel scale, and representing amplitude on a logarithmic scale.

5.2 Automatic Speech Recognition

Automatic Speech Recognition (ASR) is the task of converting spoken language into written text. Traditionally, this problem was tackled using hand-engineered features and statistical models:

- *Feature extraction*: audio signals were first transformed into Mel-frequency cepstral coefficients (MFCCs) derived from Mel spectrograms.
- *Modeling*: these features were then fed into Hidden Markov Models (HMMs) combined with Gaussian Mixture Models (GMMs) for sequence prediction.

However, the advent of DL has significantly changed the landscape of ASR.

A common approach using Convolutional Neural Networks (CNNs) involves classifying phonemes from raw audio or Mel spectrograms. This method, however, introduces several issues:

- CNNs make predictions over fixed-size windows, but phonemes and words vary in duration.
- There's a need to determine how many windows correspond to each linguistic unit.

These limitations highlight the need for sequence models that can align input and output of different lengths. A breakthrough in ASR came with encoder-decoder architectures, which can handle variable-length input and output sequences and learn temporal alignments without requiring fixed-size segmentation.

Wav2vec Wav2Vec represents a significant advancement in ASR. It is a transformer-based model that operates directly on raw audio waveforms. A convolutional frontend first extracts latent representations from the audio, which are then fed into a transformer encoder to capture temporal dependencies and contextual information. Importantly, Wav2Vec is trained in a self-supervised manner, enabling the use of large, unlabeled audio datasets.

Whisper More recently, the Whisper model has achieved state-of-the-art performance in ASR. Unlike Wav2Vec, Whisper uses Mel spectrograms as its input representation and adopts an architecture inspired by Vision Transformers. Its training paradigm is weakly supervised and leverages vast amounts of diverse, multilingual data. This approach supports multiple languages and tasks, such as transcription, translation, and language detection, within a single unified model.

5.2.1 Evaluation

The performance of ASR systems is typically evaluated using two main metrics: Word Error Rate (WER) and Sentence Error Rate (SER). WER quantifies the number of word-level substitutions, deletions, and insertions needed to transform the predicted transcription into the correct one, normalized by the number of words in the reference. SER, on the other hand, measures the proportion of sentences that contain at least one error, offering a complementary view of system performance at the sentence level.

5.2.2 Advanced Automatic Speech Recognition

Beyond general-purpose transcription, ASR systems can be adapted to improve accuracy for individual speakers. Speaker-specific variation, such as vocal tract length and pitch, can significantly affect recognition performance. Techniques such as vocal tract length normalization, which warps the frequency axis of the speech spectrum, help account for anatomical differences between speakers. Similarly, pitch normalization—particularly useful for recognizing children's speech. Another approach involves adapting a pre-trained acoustic model using a small amount of data from a new speaker to fine-tune the system for improved personalization.

An additional challenge in ASR is handling non-verbal and non-word sounds, which frequently occur in real-world audio. These include sounds like coughing, sighing, and environmental noises such as telephone rings or door slams. To accommodate these, special phonetic units can be defined, and corresponding placeholder words added to the model's lexicon and language model. Training data must include annotations for these special tokens to ensure the model can recognize and appropriately represent such sounds in the transcript.

5.3 Speech Synthesis

Text-to-speech (TTS) systems, much like their speech-to-text counterparts, have undergone significant advancements in recent years. The goal of TTS is to convert a given text string into a natural-sounding audio waveform. Modern TTS systems are typically implemented as a three-stage pipeline:

1. Maps input text to a phoneme sequence, accounting for pronunciation rules.
2. Converts the phonemes into an acoustic representation, usually a Mel spectrogram, which encodes frequency and time information.
3. Generates a raw audio signal from the spectrogram, effectively synthesizing speech.

One of the challenges in this pipeline is the expansion of abbreviated or irregular text into a fully verbalized form—a process known as normalization. This task often requires contextual understanding. Another related problem is homograph disambiguation. English contains many homographs—words that are spelled identically but pronounced differently depending on meaning. Determining the correct pronunciation thus requires contextual cues, further complicating the synthesis task.

Tacotron 2 A well-known example of a modern TTS system is Tacotron 2, introduced in 2018. This architecture employs an LSTM-based encoder-decoder to generate a Mel spectrogram from the input text. It produces high-quality spectrograms that are then fed into a separate vocoder to synthesize the audio waveform.

The vocoder used in Tacotron 2 is WaveNet, a generative model developed by DeepMind. WaveNet transforms the Mel spectrogram into an audio waveform using an auto-regressive approach based on dilated convolutions. These dilated convolutions enable the model to capture long-range temporal dependencies in the audio signal by expanding the receptive field without increasing the number of layers excessively. WaveNet produces highly natural and expressive speech, though its auto-regressive nature makes real-time synthesis computationally intensive.

5.3.1 Evaluation

Evaluating the performance of a speech synthesis system generally requires subjective human judgment. Two primary criteria are assessed: intelligibility and quality. Intelligibility refers to the listener's ability to correctly understand and interpret the spoken content, including fine-grained phonetic distinctions. Quality refers to how natural, fluent, and clear the synthesized speech sounds.

Evaluation methods include the Mean Opinion Score (MOS), where human raters score utterances on a scale from 1 to 5 based on overall quality. Another approach is the AB test, in which listeners are presented with the same utterance synthesized by two different systems and asked to select the one they prefer. This process is repeated across a set of utterances (typically around 50) to gather comparative performance data.

CHAPTER 6

Chatbots

6.1 Spoken dialog

In spoken interaction, participants alternate turns, with each turn ranging from a single word to multiple sentences. Turn-taking is fluid and often negotiated dynamically between speakers.

6.1.1 Conversational analysis

Speech acts represent the functional intention behind utterances. Common categories include:

- *Constatives*: commit the speaker to a belief or claim (e.g., answering, confirming, denying).
- *Directives*: aim to influence the behavior of the addressee (e.g., requesting, advising, inviting).
- *Commissives*: commit the speaker to future action (e.g., promising, planning, opposing).
- *Acknowledgments*: express social or interactional responses (e.g., thanking, apologizing, greeting).

6.2 Conversational agents

Conversational agents, also known as dialogue systems, chatbots, or voice interfaces, are AI-driven systems designed to interact with humans through natural language. They serve diverse purposes, including voice-controlled automation, entertainment, therapeutic application and service access. They can be classified as:

- *Open-domain chatbots*: designed for unstructured, human-like conversations.
- *Task-oriented dialogue systems*: focused on completing specific tasks.

6.2.1 Open domain chatbots

6.2.1.1 Rule-based

Rule-based chatbots rely on a predefined set of pattern-action rules to generate responses. Early examples include ELIZA (1966), which mimicked a Rogerian psychotherapist, and PARRY (1971), which simulated a patient with paranoid schizophrenia. These systems did not require real-world knowledge but instead used linguistic and psychological cues to sustain conversations.

Eliza ELIZA gave the impression of linguistic coherence by reflecting the user's statements back to them, simulating the behavior of a Rogerian therapist. This approach required minimal understanding of the world. ELIZA's architecture was composed of simple pattern-matching rules triggered by specific keywords. Each keyword was associated with a set of transformation patterns to generate responses. If a user input matched multiple patterns, ELIZA would select the most specific rule. In cases where no keywords were matched, it defaulted to a generic, non-committal response. Despite its simplicity, ELIZA could maintain seemingly coherent conversations and recall references made earlier in the dialogue, contributing to its perceived intelligence.

Parry PARRY extended ELIZA's design by incorporating a rudimentary mental model to simulate paranoid behavior. Like ELIZA, it used pattern-action rules, but it added a richer control structure and internal state modeling. PARRY tracked psychological variables such as anger, fear, and mistrust, all initially set to low levels. These variables were dynamically updated based on user input. The chatbot's responses were influenced by its current mental state, making its behavior appear more contextually sensitive and emotionally driven.

6.2.1.2 Corpus-based

Modern corpus-based chatbots rely heavily on large-scale conversational datasets. These systems raise important concerns around privacy, particularly regarding the need to remove personally identifiable information from training data.

Corpus-based approaches fall into two main categories: retrieval-based and generation-based methods:

- *Retrieval-based systems*: in retrieval-based systems, the chatbot selects an appropriate response from a pre-existing corpus. Given a user query, the system identifies the most similar conversational context and retrieves the corresponding reply.

- *Generation-based systems*: generative models, on the other hand, synthesize responses word-by-word, conditioned on the input query and, optionally, the dialogue history. These models typically use encoder-decoder or decoder-only neural architectures trained on large volumes of conversational data. Despite their flexibility, generative chatbots often suffer from producing dull, repetitive, or overly generic responses, which can prematurely end conversations.

A hybrid approach combines retrieval with generation: retrieving a candidate response and then refining it via a generative model. This is particularly effective for task-oriented domains where responses can be more scripted and constrained.

While corpus-based chatbots can give the illusion of understanding, they do not possess true comprehension. Their responses are shaped entirely by patterns in the training data. This can lead to ethical and functional concerns—especially if users assume a deeper understanding than actually exists. Rule-based systems, although interpretable, are labor-intensive and brittle. Retrieval-based systems, meanwhile, are limited to reusing information seen during training, restricting their ability to generalize.

6.2.2 Task-oriented dialog agents

Task-oriented dialog agents are designed to assist users in completing specific goals, such as booking a ticket, ordering food, or finding information. These systems typically follow a goal-oriented framework and often employ a frame-based architecture, in which user intents are represented as frames consisting of slots that need to be filled with relevant information.

6.2.2.1 Frame-based

One of the earliest examples of a frame-based system is GUS (1977). It introduced a modular architecture in which each frame represented a specific action or task and contained multiple slots. Each slot was associated with a question the system could ask the user to elicit the required information.

The dialogue proceeded by the system asking questions and populating any slots that the user explicitly filled. Once all required slots were completed, the system could execute a database query or perform the corresponding action. GUS typically supported multiple frames, and the system had to determine which frame and which slot the user's utterance corresponded to, switching control accordingly. This process relied on condition-action rules.

The Natural Language Understanding (NLU) component in GUS extracted three key elements from user input:

1. *Domain classification*: identifying the topic area of the user's request.

2. *Intent recognition*: determining the user's goal.

3. *Slot filling*: extracting values for relevant slots.

Slot-filling was rule-based, and responses were generated using predefined templates. This rule-and-template paradigm remains common in many industry applications due to its interpretability and control.

6.2.2.2 Dialog state

Modern task-oriented systems are typically modular and consist of the following components:

1. *Natural Language Understanding* (NLU): uses machine learning techniques to extract slot values and user intent from natural language input.

2. *Dialogue State Tracker* (DST): maintains a representation of the current dialogue state, which includes the user's latest intent, filled slots, and constraints. This component must track user goals across multiple turns, including corrections and updates.

3. *Dialogue Policy*: determines the next system action based on the dialogue state. Early systems used fixed policies that asked questions until the frame was filled. More advanced systems incorporate heuristics or learned policies that can decide when to ask clarification questions, confirm ambiguous input, or take other context-sensitive actions.

4. *Natural Language Generation* (NLG): converts system actions into natural-sounding utterances. While early systems relied on templates, newer approaches use neural models to produce more varied and human-like responses.

To manage conversations effectively, systems map user inputs to dialogue acts—abstract representations of communicative functions. Dialogue act tagging helps structure conversations and track mutual understanding (or grounding).

Slot filling can be approached using: classifiers and sequence labelers that label tokens in an utterance and end-to-end sequence-to-sequence models that map entire user inputs to structured representations. Systems must detect the domain and user intent to route the input to the appropriate frame or service. Dialogue state tracking is often handled by models like the Neural Belief Tracker, which updates the state based on user input in a data-driven manner. Importantly, systems must handle corrections, which occur when users modify or repeat themselves after a misunderstanding. Detecting and responding appropriately to these correction acts remains a challenging problem, as users may rephrase, negate previous answers, or express frustration implicitly.

While frame-based systems are effective for well-defined tasks, they can be brittle, expensive to develop, and difficult to scale across domains. Incorporating more flexible learning-based methods—particularly for intent recognition, slot filling, and state tracking—remains an active area of research and industrial innovation.

6.3 Dialog policy and generation

Effective dialogue systems must decide what to say and how to say it at each turn of interaction. This involves both high-level dialogue policy (choosing the system action) and Natural Language Generation.

At each turn, the system must choose the next action based on either the entire conversation history, or the current dialogue state.

Dialogue systems inevitably make errors due to noise in speech recognition, ambiguous user input, or misunderstanding of intent. To manage this, they use two key mechanisms confirming their understanding with the user and rejecting or flagging input they don't understand. Confirmation can be: explicit (the system repeats back what it believes it understood) or implicit (the system proceeds as if the input was understood). Rejection is used when the system cannot interpret the input confidently Systems may use progressive prompting: after repeated misunderstandings, they offer more specific guidance instead of repeating the same question. Modern systems often base decisions on confidence scores from the ASR (Automatic Speech Recognition) or NLU (Natural Language Understanding) components. Confidence scores reflect how likely the system believes its interpretation is correct. Based on these scores, the system decides whether to confirm, reject, or proceed.

6.3.1 Natural Language Generation

In information-state architectures, NLG is typically a two-stage process:

- *Content planning*: the dialogue policy selects the speech act and the content to convey.
- *Sentence realization*: the system generates fluent natural language based on the planned content—either answering a question, confirming a detail, or prompting the user.

6.3.1.1 Sentence realization

Generating high-quality, context-sensitive language is challenging—especially given sparse training data. Many domain-specific values may be rarely or never seen in training. To generalize better, NLG systems use delexicalization: slot values in training data are replaced with generic placeholders and then systems are trained to generate delexicalized templates from input frames. Then, in a post-processing step, the output is relexicalized by inserting the correct slot values. Modern systems often use encoder-decoder architectures to map structured frames to delexicalized sentences.

6.3.2 Clarification questions

Clarification is essential when part of the user's utterance is misunderstood. Methods for generating clarification questions include rule-based templates and ML classifiers.

6.4 Dialog systems evaluation

Evaluating dialogue systems remains a complex and evolving challenge. Unlike task-specific models, open-domain conversational agents must be assessed across multiple dimensions of quality—many of which are inherently subjective and difficult to measure automatically. Eight commonly considered dimensions of quality in dialogue include: repetition, interestingness, coherence, fluency, listening, inquisitiveness, humanness, and engagingness.

6.4.1 Human evaluation

Human assessments are essential for gauging subjective aspects of quality. These evaluations are typically divided into two formats:

- *Participant evaluation*: users interact with the system and then answer targeted questions.
- *Observers evaluation*: annotators compare pairs of conversations and assess them along qualitative axes.

These comparative judgments often provide more reliable insight than absolute scoring.

6.4.2 Automatic evaluation

Automatic evaluation of dialogue systems remains an open research problem. Traditional metrics from fields like machine translation are rarely used in conversational AI because they show low correlation with human judgments, and fail to capture contextual appropriateness, tone, or relevance in conversation. The main alternative approaches are:

- *Adversarial evaluation*: inspired by the Turing Test, this method trains a classifier to distinguish between human and machine responses. The more the model can fool the classifier, the better its perceived quality.
- *LLM-as-a-judge*: these models are prompted to score or rank conversations based on quality dimensions—offering scalable, semi-automated evaluation with surprisingly strong alignment to human preferences.

For systems built to accomplish specific tasks evaluation focuses more on functional success such as: end-to-end task success, slot error rate, user studies, efficiency, and system robustness.

6.5 Ethics

Ethical considerations are foundational in the design and deployment of artificial agents. From early fictional warnings such as Mary Shelley's Frankenstein, which depicted the dangers of creating intelligent beings without regard for moral responsibility, to modern real-world failures, the lesson is clear: neglecting ethical principles can have serious consequences.

Key areas of concern include:

- *Safety*: preventing harm caused by inappropriate or dangerous behavior.
- *Representational harm*: avoiding reinforcement of stereotypes or marginalization of social groups.
- *Privacy*: safeguarding users' personal data from leaks or misuse.

6.5.1 Safety

Ensuring the safety of users is paramount. Mental health chatbots must exercise extreme caution. A poorly worded response can have serious emotional consequences. In-vehicle conversational agents must remain context-aware to avoid distracting the driver or compromising road safety. Systems interacting with humans must be designed with rigorous guardrails to prevent unintended harm, especially when users may place implicit trust in the agent's responses.

6.5.2 Representational harm

If the training data contains biased or harmful content, the system is likely to reproduce it. AI systems can reinforce and even normalize the harmful biases present in the datasets they learn from. As a result, significant research efforts are now directed toward detecting, mitigating, and removing biased or toxic content from training data.

6.5.3 Privacy

Privacy concerns arise from both accidental and intentional information leakage. Designing privacy-preserving dialogue systems is a critical challenge moving forward. Developers must prioritize data minimization, local processing, and transparent data policies to earn and maintain user trust.

CHAPTER 7

Agentic Artificial Intelligence

7.1 Introduction

Agentic AI Design Patterns refer to commonly used architectural strategies for deploying Large Language Models (LLMs) in autonomous or semi-autonomous workflows. These patterns enable LLMs to reason, act, and collaborate in ways that mirror intelligent agent behavior. The following are some of the most widely adopted patterns:

- *Reflection pattern*: the LLM evaluates and critiques its own outputs. This can be used to enforce behavioral constraints.
- *Tool use pattern*: the LLM interacts with external tools and incorporates their outputs into its reasoning. Tools may include calculators, search engines, or APIs, enabling the model to go beyond its static knowledge.
- *Reasoning and acting pattern*: the LLM alternates between reasoning steps and tool use to iteratively achieve a goal.
- *Planning pattern*: the LLM creates a multi-step plan to accomplish a complex task. It monitors the execution of subtasks, handles failures, and adjusts the plan dynamically as needed to ensure the desired outcome.
- *Multi-agent pattern*: multiple LLMs function as distinct agents, each with specialized roles or capabilities. These agents communicate and collaborate to solve problems collectively, often outperforming a single-agent approach for complex tasks.

7.1.1 LangChain

LangChain is an open-source framework designed to streamline the development of LLM-powered applications. LangChain supports modular design, making it easy to integrate tools, prompts, and memory.

7.2 Usage

LLMs can be designed to interact with external tools by following a few key mechanisms:

- *Instruction tuning*: modern LLMs are often fine-tuned with instructions that teach them how to use tools. During training, they are exposed to examples that demonstrate tool invocation and response integration.

- *Declaring available tools*: at the beginning of an interaction, the available tools can be declared explicitly. The LLM is informed of each tool's purpose and usage pattern, enabling it to make appropriate decisions during the conversation.

- *Specialized syntax for tool use*: interactions with tools are formatted using special tokens or syntactic conventions, allowing the LLM to differentiate between user inputs, tool calls, and tool outputs. Tools are treated as participants in the dialogue, enabling a seamless multi-turn workflow.

- *Backend parsing and execution*: while the LLM generates text to indicate a tool call, the underlying system interprets this output, executes the corresponding tool or API call, and feeds the result back into the conversation.

7.3 Retrieval Augmented Generative models

Building a high-performing RAG system involves a series of nuanced design decisions that significantly affect system accuracy, latency, and scalability. Each of these decisions involves trade-offs between accuracy, retrieval speed, memory consumption, and interpretability.

Evaluating RAG performance is challenging due to the inherently generative nature of the output. Traditional metrics may not suffice, especially when the exact match is unreliable. Generated outputs may use different wording than ground-truth answers while still conveying the same meaning. To address this, researchers often employ LLM-based evaluation strategies:

- *LLM-as-a-Judge*: a single LLM is tasked with comparing the generated response against the expected answer to determine semantic equivalence.

- *LLM-as-a-Jury*: multiple LLMs independently assess the similarity between the generated output and the reference, then vote on agreement. This ensemble approach increases robustness and mitigates individual model bias.