



307307

Introduction to Text Mining and Natural Language Processing

What is Natural Language Processing?

- Natural Language Processing (NLP) is a field at the intersection of computer science, artificial intelligence, and linguistics focused on enabling computers to understand, interpret, and generate human language.
- It forms the foundation for all language technologies we use today, from search engines to voice assistants and modern large language models.

Historical Development of NLP

NLP has evolved dramatically over the decades:

1950s-1960s: Rule-Based Approaches

- Focus on machine translation during the Cold War
- Primitive rule-based systems with hand-crafted grammars
- Limited by the complexity of language rules

1970s-1980s: Statistical Revolution Begins

- Introduction of probabilistic models
- Hidden Markov Models for part-of-speech tagging
- Still limited by computational power and data availability

1990s-2000s: Statistical NLP Flourishes

- Large text corpora become available
- Machine learning approaches gain prominence
- Statistical machine translation emerges

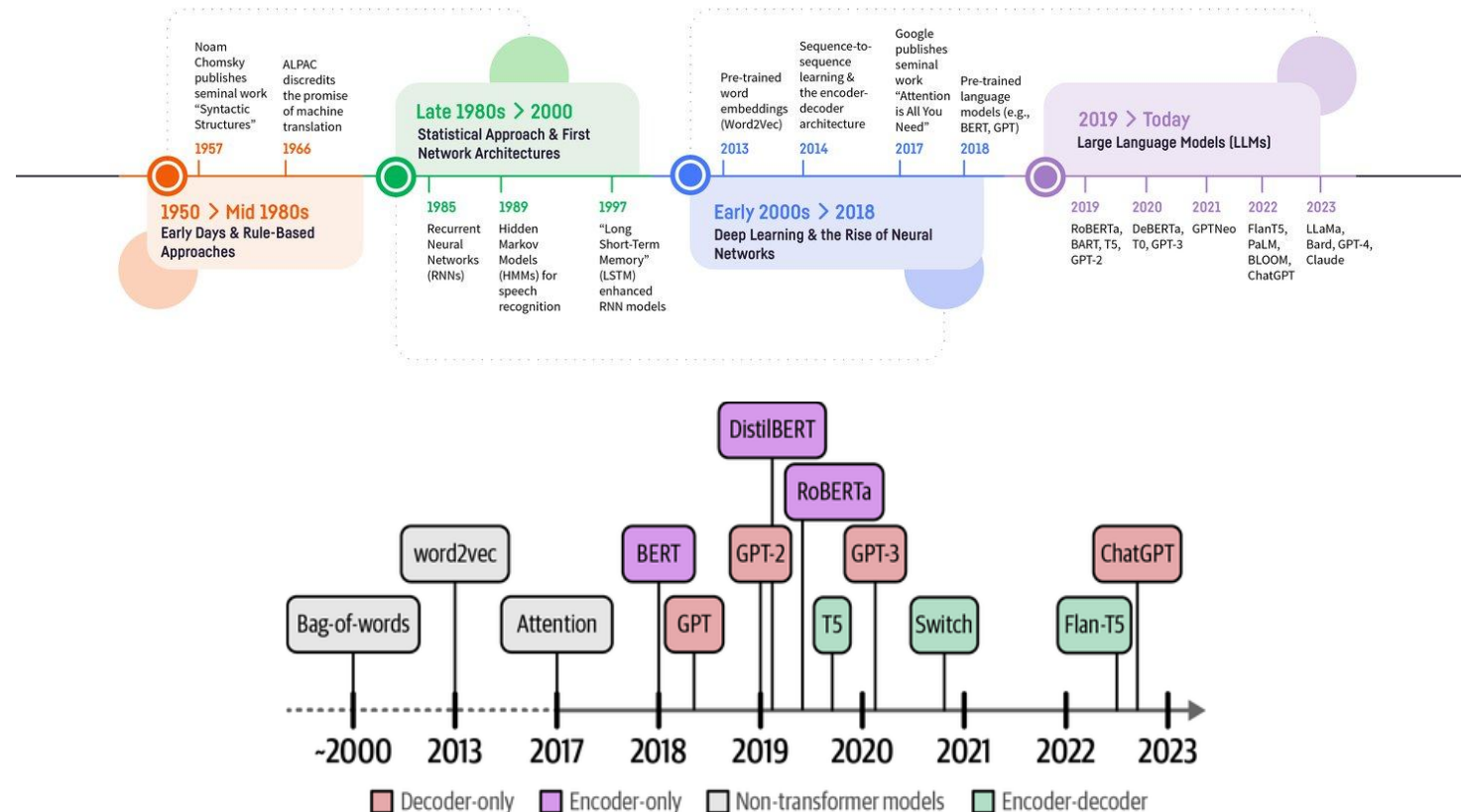
2010-2018: Neural Revolution

- Word embeddings (Word2Vec, GloVe) transform representation
- Recurrent Neural Networks and LSTMs for sequence modeling
- Attention mechanisms improve performance

2018-Present: Transformer Era

- BERT, GPT, and other transformer models revolutionize the field
- Pre-training and fine-tuning paradigm
- Increasingly human-like language understanding capabilities

The History of NLP



Why is NLP Challenging?

Human language is ambiguous: Words and sentences can have multiple interpretations

- **Lexical ambiguity:** When a word has multiple meanings
 - Example: "The bank is closed" (financial institution or riverbank)
 - Example: "The suit was expensive" (Formal set of clothes or Legal proceeding)
 - Computational challenge: Systems must select the correct word sense from multiple possibilities
- **Syntactic ambiguity:** When a sentence can be parsed in multiple ways
 - Example: "I saw the man with the telescope" (Who has the telescope?)
 - Example: "Flying planes can be dangerous" (The act of flying planes or planes that are flying?)
 - Computational challenge: Models must determine the correct grammatical structure
- **Semantic ambiguity:** When the meaning of a sentence has multiple interpretations
 - Example: "The chicken is ready to eat" (Ready to be eaten or ready to consume food?)
 - Example: "John and Mary got married last year" (To each other or to other people?)
 - Computational challenge: Systems must infer the intended meaning based on context
- **Pragmatic ambiguity:** When the intended meaning depends on context, intent, or implied information
 - Example: "It's cold in here" (Statement of fact or request to close a window/turn up heat?)
 - Example: "Do you know what time it is?" (Yes/no question or request for the time?)
 - Computational challenge: Models must understand communicative intent beyond literal meaning

Common NLP Applications

1. Sentiment analysis: Determining the emotional tone of text
 - Identifying if customer reviews are positive, negative, or neutral
 - Tracking public opinion on products, services, or topics
2. Machine translation: Translating text between languages
 - Converting text from one language to another while preserving meaning
 - Handling cultural and linguistic differences across languages
3. Text generation: Creating human-like text
 - Automatic summarization of longer documents
 - Creative writing assistance and content creation
 - Dialogue systems for conversational AI
4. Question answering: Providing answers to natural language questions
 - Extracting answers from documents or knowledge bases
 - Understanding user intent and providing relevant information
5. Information extraction: Identifying and extracting structured information from text
 - Pulling key data points from unstructured documents
 - Converting text documents into structured databases
6. Text classification: Categorizing text into predefined categories
 - Sorting emails into spam/not spam
 - Organizing documents by topic or content type

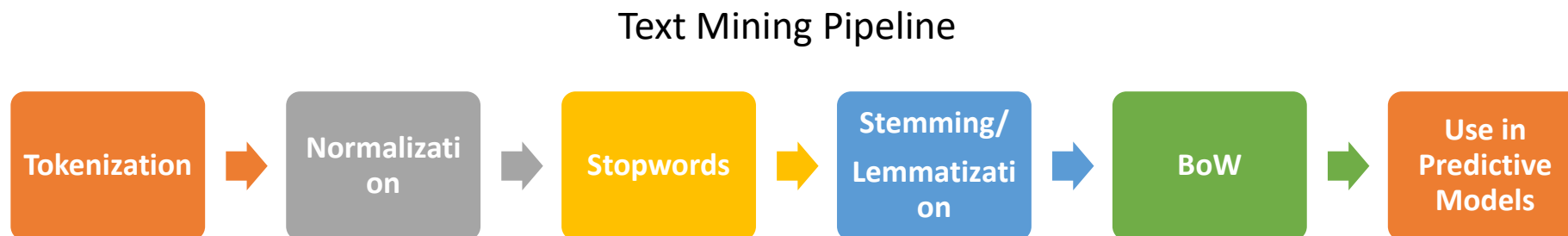
Text Processing Pipeline

Text Mining Pipeline

In this chapter, we will discuss the common methods that we can use to mine information from text. We will learn about text-processing techniques that can assist us in converting text into numerical representation that is suitable for mining information.

We will discuss the techniques and the pipeline shown below:

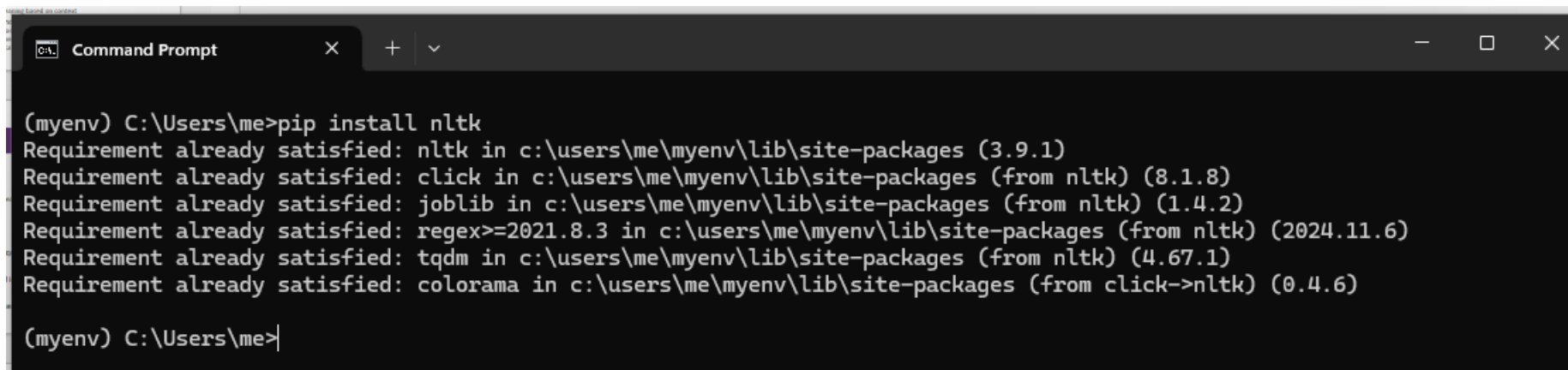
1. Sentence Tokenization
2. Text Normalization
3. Stopwords Removal
4. Stemming and Lemmatization
5. Representing text using the Bag of Words method
6. Implementing a predictive model based on text representations



Introduction to NLTK (Natural Language Toolkit)

- In this chapter we will use Python NLTK library to process text.
- NLTK is a Python text processing library that was developed in 2001. It supports many essential NLP tasks like Tokenization, Stemming and Lemmatization, Sentiment Analysis and many other features.
- We can install NLTK on our machine using the following command:

pip install nltk



```
(myenv) C:\Users\me>pip install nltk
Requirement already satisfied: nltk in c:\users\me\myenv\lib\site-packages (3.9.1)
Requirement already satisfied: click in c:\users\me\myenv\lib\site-packages (from nltk) (8.1.8)
Requirement already satisfied: joblib in c:\users\me\myenv\lib\site-packages (from nltk) (1.4.2)
Requirement already satisfied: regex>=2021.8.3 in c:\users\me\myenv\lib\site-packages (from nltk) (2024.11.6)
Requirement already satisfied: tqdm in c:\users\me\myenv\lib\site-packages (from nltk) (4.67.1)
Requirement already satisfied: colorama in c:\users\me\myenv\lib\site-packages (from click->nltk) (0.4.6)

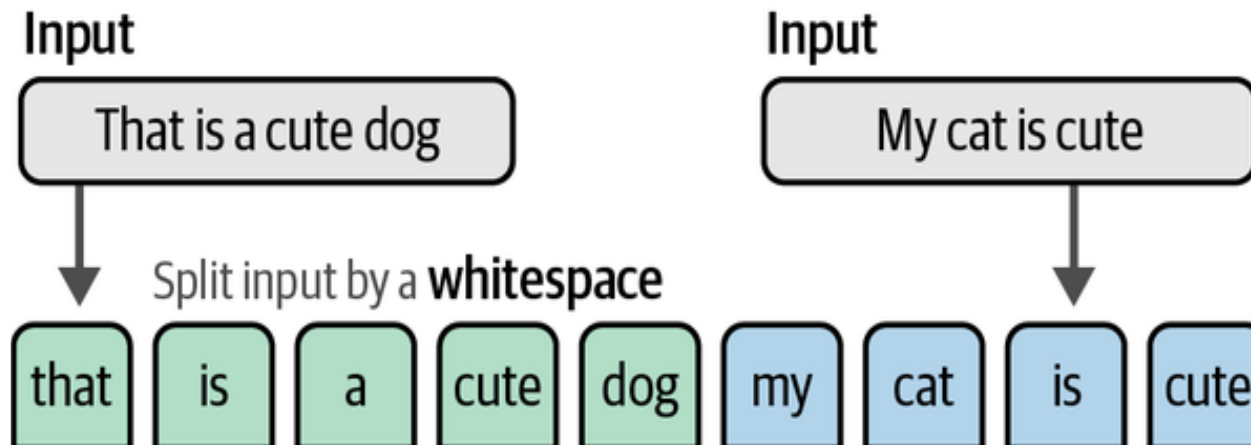
(myenv) C:\Users\me>
```


Fundamental NLP Techniques

Tokenization

Tokenization: Breaking text into words, phrases, or other meaningful elements

- Word tokenization: Splitting "I love NLP." into ["I", "love", "NLP", "."]
- Subword tokenization: Breaking words into meaningful pieces (e.g., "playing" → ["play", "##ing"])
- Character tokenization: Splitting text into individual characters



Tokenize text using NLTK package

```
from nltk.tokenize import word_tokenize

#punkt is a pretrained tokenization model used for splitting text into sentences and words.
nltk.download('punkt')

sentence = "Large language models are revolutionizing business applications."
tokens = word_tokenize(sentence)
print(tokens)
```

Output:

```
['Large', 'language', 'models', 'are', 'revolutionizing', 'business', 'applications',  
'.']  
[nltk_data] Downloading package punkt to /home/me/nltk_data... [nltk_data] Package  
punkt is already up-to-date!
```

Normalization

Normalization techniques convert text to a standard form to reduce variability.

The following code uses two techniques:

1. Converting text into lower case
2. Removing punctuation from text

```
# Lowercasing
normalized_tokens = [token.lower() for token in tokens]
print(normalized_tokens)

# Removing punctuation
import re
normalized_tokens = [re.sub(r'^\w\s', '', token.lower()) for token in tokens]
print(normalized_tokens)
```

The regular expression `r'^\w\s'` does the following:

- `r'...'`: This denotes a raw string in Python. Raw strings treat backslashes `\` as literal characters, which is crucial for regular expressions.
- `[...]`: This represents a character set. `^`: Inside a character set, `^` negates the set. It means "match any character not in this set."
- `\w`: This matches any word character, which includes alphanumeric characters (letters and digits) and the underscore (`_`).
- `\s`: This matches any whitespace character, such as spaces, tabs, and newlines.
- Therefore, `r'^\w\s'` matches any character that is not a word character or a whitespace character. In simpler terms, it matches any non-alphanumeric, non-underscore, and non-whitespace character. This effectively targets punctuation marks, symbols, and other special characters.

Output:

```
['large', 'language', 'models', 'are', 'revolutionizing',  
'business', 'applications', '.'] ['large', 'language', 'models',  
'are', 'revolutionizing', 'business', 'applications', '']
```

Stopwords Removal

- Stopwords removal process eliminates common words that add little meaning to text
- Words like (is, are, at, on, the,...etc) can be removed from text to convert text to a standard form to reduce variability

```
#nltk.download('stopwords') #stopwords are common words that are often
removed from text as they are not useful for analysis.
from nltk.corpus import stopwords

nltk.download('stopwords')
stop_words = set(stopwords.words('english'))

print("These words are removed from the text: ", stop_words)
```

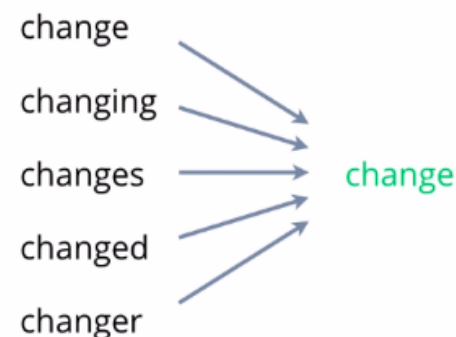
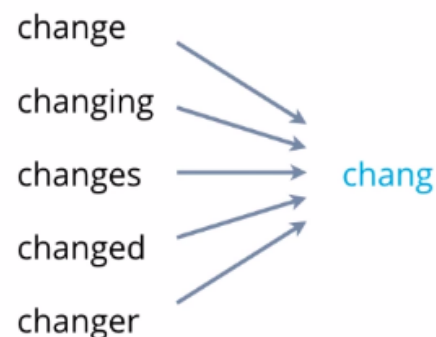
Output:

These words are removed from the text: {'during', 'off', 'each', 'own', 'nor', 'we've', 'you'll', 'ourselves', 'between', 'same', 'such', 'against', 'but', 'mustn't', 'aren't', 'wouldn't'...etc

Stemming and Lemmatization

- Stemming and lemmatization are text normalization techniques used in Natural Language Processing (NLP) to reduce words to their base or root forms.
- They help in improving search engines, text mining, and language modeling by standardizing words with similar meanings.

Stemming vs Lemmatization



Stemming

- Definition: Stemming is a rule-based process that removes prefixes and suffixes (affixes) from a word to obtain its base form, often producing words that may not be actual words.
- Method: Uses algorithms like Porter Stemmer and Snowball Stemmer.
- Example:
 - "Running" → "Run"
 - "Studies" → "Studi"
 - "Better" → "Better" (Incorrectly unchanged)
 - "Caring" → "Car"
- Limitations:
 - Often results in words that are not real words ("Studies" → "Studi").
 - Does not consider the actual meaning of words.

Lemmatization

- Definition: Lemmatization is a dictionary-based approach that reduces words to their meaningful root form (lemma), considering the word's context and part of speech.
- Method: Uses linguistic analysis and dictionaries (like WordNet Lemmatizer).
- Example:
 - "Running" → "Run"
 - "Studies" → "Study"
 - "Better" → "Good" (Correct Lemma)
 - "Caring" → "Care"
- Advantages:
 - More accurate than stemming.
 - Produces real words with proper meanings.
- Use Cases:
 - Stemming is useful when speed is more important than accuracy (e.g., search engines).
 - Lemmatization is better for semantic analysis where meaning matters (e.g., AI chatbots, NLP applications).

Stemming Example

```
from nltk.stem import PorterStemmer  
  
stemmer = PorterStemmer()  
stemmed_tokens = [stemmer.stem(token) for token in filtered_tokens]  
  
print(stemmed_tokens)
```

Lemmatizing Example

```
#nltk.download('wordnet')  
  
from nltk.stem import WordNetLemmatizer  
  
lemmatizer = WordNetLemmatizer()  
lemmatized_tokens = [lemmatizer.lemmatize(token) for token in filtered_tokens]  
  
print(lemmatized_tokens)
```

Text Representation

Converting Text into Numbers

Bag of Words Method

Bag of Words is a fundamental technique in Natural Language Processing (NLP) that represents text as a collection of word frequencies, disregarding grammar and word order.

Core Concept

- The Bag of Words model converts text documents into numerical feature vectors by:
 - Creating a vocabulary of all unique words in the corpus
 - For each document, counting how many times each vocabulary word appears
- Simple Example:
 - Consider these two short sentences:
 - "I love natural language processing"
 - "Processing natural language is fun"
 - The vocabulary would be:
 - {"I", "love", "natural", "language", "processing", "is", "fun"}
 - Document 1 vector: [1, 1, 1, 1, 1, 0, 0]
 - Document 2 vector: [0, 0, 1, 1, 1, 1, 1]

Raw Text

it is a puppy and it
is extremely cute

Bag-of-words vector

it	2
they	0
puppy	1
and	1
cat	0
aardvark	0
cute	1
extremely	1
...	...

BoW Basic Term Frequency Example

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
    "Large language models revolutionize business.",
    "Business applications benefit from AI.",
    "Language models learn from text data."
]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)

print(vectorizer.get_feature_names_out())
print(X.toarray())
```

	ai	applicatio ns	benefit	business	data	from	language	large	learn	models	revolutio nize	text
Vector 1	0	0	0	1	0	0	1	1	0	1	1	0
Vector 2	1	1	1	0	0	0	0	0	0	0	0	0
Vector 3	0	0	0	0	1	1	1	0	1	1		

BoW Method

Applications

- BOW is commonly used in:
- Text classification (spam detection, sentiment analysis)
- Document clustering
- Information retrieval
- Topic modeling

Advantages

- Simple to understand and implement
- Computationally efficient
- Effective for many basic NLP tasks

Limitations

- Loses word order information
- Ignores semantics and context
- Creates high-dimensional, sparse vectors
- Struggles with out-of-vocabulary words

TF-IDF (Term Frequency-Inverse Document Frequency) Method

- TF-IDF is a statistical measure that evaluates the importance of a word in a document relative to a collection of documents (corpus).
- It improves upon the basic Bag of Words model by weighing terms based on how unique they are to a specific document.

$$w_{i,j} = tf_{i,j} \times \log \left(\frac{N}{df_i} \right)$$

tf_{ij} = number of occurrences of i in j

df_i = number of documents containing i

N = total number of documents

Key Idea:

- TF-IDF increases when a word appears frequently in a document but rarely across the corpus, making it a good indicator of relevance.

TF/IDF Example: Compute the TD/IDF for the words learn and data in sentence 1

We have two documents, document 1 and document 2

- Document 1: AI models learn from data
- Document 2: Data drives business decisions

1. Compute Term Frequency (TF) = Occurrences of word in document / Total words in document

Word	Sentence 1 (5 words)	Sentence 2 (4 words)
data	$1 / 5 = 0.2$	$1 / 4 = 0.25$
learn	$1 / 5 = 0.2$	$0 / 4 = 0$

Weights a word in a single document

2. Compute Inverse Document Frequency (IDF) = $\log(\text{Count of all Documents} / \text{Count of Documents Containing Word})$

Word	Count of Documents Containing Word	Count of all Documents	IDF
data	2	2	$\log(2/2) = 0$
learn	1		$\log(2/1) \approx 0.693$

Weights a word across all documents (General Index)

3. Compute TF/IDF = TF x IDF

Word	Document 1 TF	IDF	TF-IDF
data	0.2	0	$0.2 \times 0 = 0.0$
learn	0.2	0.693	$0.2 \times 0.693 \approx 0.1386$

data is more common across documents \rightarrow TF-IDF = 0 (less informative)

learn is unique to document 1 \rightarrow Higher TF-IDF (more informative)

TF-IDF (Term Frequency-Inverse Document Frequency)

```
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
    "Large language models revolutionize business.",
    "Business applications benefit from AI.",
    "Language models learn from text data."
]

tfidf_vectorizer = TfidfVectorizer()

# Transform the corpus into a document-term matrix
# Each row represents a document in the corpus
# Each column represents a unique word in the corpus
X_tfidf = tfidf_vectorizer.fit_transform(corpus)

df = pd.DataFrame(X_tfidf.toarray(), columns=tfidf_vectorizer.get_feature_names_out())
df
```

	ai	applications	benefit	business	data	from	language	large	learn	models	revolutionize	text
Doc 1	0	0	0	0.3935	0	0	0.3935	0.5174	0	0.3935	0.5174	0
Doc 2	0.4905	0.4905	0.4905	0.373	0	0.373	0	0	0	0	0	0
Doc 3	0	0	0	0	0.4595	0.3495	0.3495	0	0.4595	0.3495	0	0.4595

Advantages of TF-IDF

Advantages

- Reduces the impact of frequently occurring terms (like "the", "is", "of")
- Highlights terms that are distinctive to specific documents
- Relatively simple to implement and computationally efficient
- Works well for sparse data

Limitations

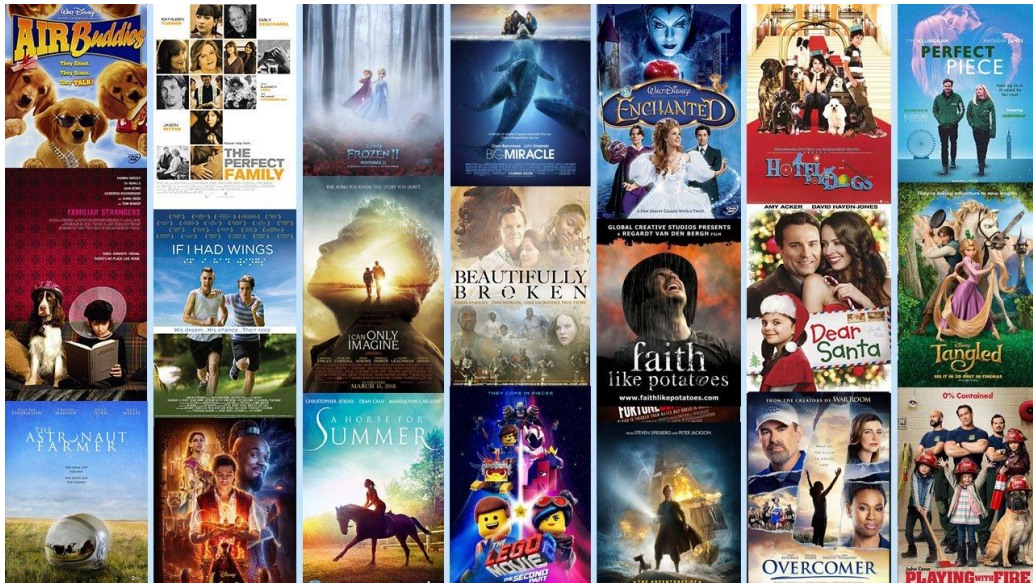
- Still ignores word order and semantics
- Cannot capture relationships between related terms
- May not work well for very short texts
- Struggles with synonyms and polysemy
- TF-IDF serves as a cornerstone technique in information retrieval and text mining, and remains widely used despite the emergence of more advanced embedding-based approaches.

Practical Example

Movie Review Sentiment Analysis Pipeline

Sentiment Analysis of Movie Reviews

- This practical example will go through the entire text preprocessing pipeline and show how it contributes to a real-world NLP task.
- Open the attached Jupyter Notebook to review the example.
- We will use a simple corpus, the user reviews are registered as 0 or 1 → Like, Don't Like.
- We can also use the same text processing pipeline on IMDB dataset.



Naive Bayes Classifier for Sentiment Analysis

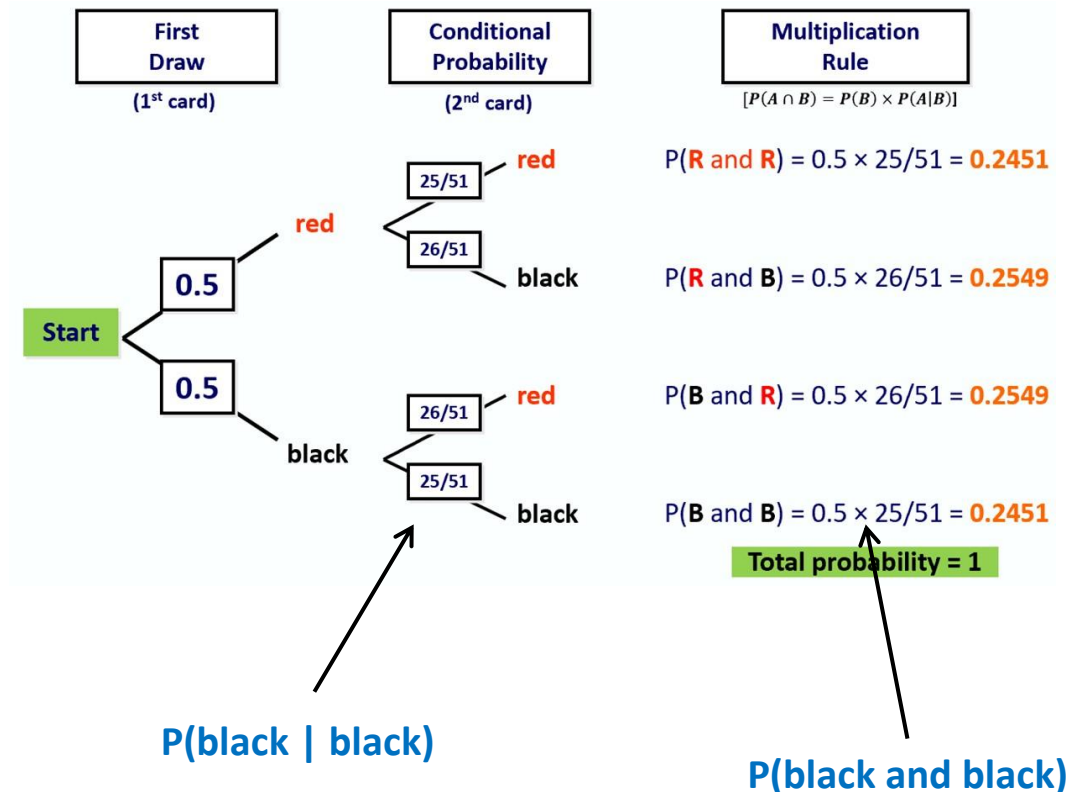
- A probabilistic machine learning model based on Bayes' Theorem
- Assumes independence between features (words in our case)
- Commonly used for text classification (e.g. spam detection, sentiment analysis)

Bayes' Theorem (General Form)

$$P(A | B) = \frac{P(B | A) \cdot P(A)}{P(B)}$$

Where:

- $P(A | B)$ is the probability of event A given B
- $P(B | A)$ is the likelihood of event B given A
- $P(A)$ is the prior probability of A
- $P(B)$ is the marginal probability of B



Bayes vs. Naïve Bayes

Bayes (non-naive)

Bayes' theorem (without the **naïve** assumption) would treat the sentence as a **single unit**, considering the **joint probability of the whole sentence** under each class:

$$P(\text{Positive} \mid \text{"I love this movie"}) = \frac{P(\text{"I love this movie"} \mid \text{Positive}) \cdot P(\text{Positive})}{P(\text{"I love this movie"})}$$

But estimating $P(\text{"I love this movie"} \mid \text{Positive})$ is **not feasible** unless the exact sentence has occurred frequently in the training data. That's rare in most practical datasets. So:

- **Challenge:** Requires massive training data with exact phrase repetition
- **High dimensionality & sparsity**
- **Impractical without strong language modeling**

Naive Bayes

Naive Bayes simplifies the problem by assuming **conditional independence of words** given the class:

$$P(\text{Positive} \mid \text{"I love this movie"}) \propto P(\text{Positive}) \cdot P(\text{"I"} \mid \text{Positive}) \cdot P(\text{"love"} \mid \text{Positive}) \cdot P(\text{"this"} \mid \text{Positive}) \cdot P(\text{"movie"} \mid \text{Positive})$$

Each word is treated as an independent feature.

- We multiply the conditional probabilities of each word given the class
- Apply Laplace smoothing to avoid zero-probabilities
- Compare scores across all classes and choose the one with the highest value

Naive Bayes Classifier for Sentiment Analysis

Bayes' Theorem for Text Classification

$$P(\text{Class} \mid \text{Text}) = \frac{P(\text{Text} \mid \text{Class}) \cdot P(\text{Class})}{P(\text{Text})}$$

Where:

- $P(\text{Class} \mid \text{Text})$: Probability that the given text belongs to a class (e.g., positive/negative)
- $P(\text{Text} \mid \text{Class})$: Probability of observing the text assuming a specific class
- $P(\text{Class})$: Prior probability of the class
- $P(\text{Text})$: Marginal probability of the text (can be omitted for comparison between classes)

$$P(\text{Positive} \mid \text{"great"})$$

Using Bayes' Theorem:

$$P(\text{Positive} \mid \text{"great"}) = \frac{P(\text{"great"} \mid \text{Positive}) \cdot P(\text{Positive})}{P(\text{"great"})}$$

Let's plug in from your table:

- $P(\text{"great"} \mid \text{Positive}) = \frac{15}{50}$
- $P(\text{Positive}) = \frac{50}{104}$
- $P(\text{"great"}) = \frac{18}{104}$

Now apply:

$$P(\text{Positive} \mid \text{"great"}) = \frac{(15/50) \cdot (50/104)}{18/104} = \frac{15}{18} \approx 0.833$$

Word	Positive	Negative	Row Total
love	20	2	22
great	15	3	18
amazing	12	1	13
boring	1	14	15
terrible	0	18	18
bad	2	16	18
Total	50	54	104

$$P(\text{Positive} \cap \text{"great"}) = P(\text{"great"} \mid \text{Positive}) \cdot P(\text{Positive})$$

From the Table:

- $\text{Count}(\text{"great"} \mid \text{Positive}) = 15$
- Total words in Positive reviews = 50

$$P(\text{"great"} \mid \text{Positive}) = \frac{15}{50} = 0.3$$

- Total words (Positive + Negative) = 104

$$P(\text{Positive}) = \frac{50}{104} \approx 0.481$$

Therefore:

$$P(\text{Positive} \cap \text{"great"}) = \frac{15}{50} \cdot \frac{50}{104} = \frac{15}{104} \approx 0.144$$

Problem Example

Task: Classify the sentiment of a sentence

Training Data:

1. I love this movie (Positive)
2. This film is amazing (Positive)
3. I hate this movie (Negative)
4. This film is terrible (Negative)

Classify: 'I love this film'

Step 1 – Vocabulary

Create a vocabulary of all words:

[I, love, this, movie, film, is, amazing, hate, terrible]

Step 2 – Word Frequencies

Count how often each word appears in each class:

Word	Positive	Negative
I	1	1
love	1	0
this	2	2
movie	1	1
film	1	1
is	1	1
amazing	1	0
hate	0	1
terrible	0	1

Step 3 – Class Probabilities

We have 2 Positive and 2 Negative examples.

$$P(\text{Positive}) = 2/4 = 0.5$$

$$P(\text{Negative}) = 2/4 = 0.5$$

Step 4 – Laplace Smoothing

Use **Laplace Smoothing** to handle unseen words.

Vocabulary size $V = 9$

Total words in Positive: 8

Total words in Negative: 8

Formula:

$$P(word|class) = \frac{count(word) + 1}{total_words + V}$$

Step 5 – Calculate Probabilities (Positive)

Classify: "I love this film"

Positive:

$$\begin{aligned} &P(Positive) \times P(I|Positive) \times P(love|Positive) \times P(this|Positive) \times P(film|Positive) \\ &= 0.5 \times \frac{1+1}{8+9} \times \frac{1+1}{8+9} \times \frac{2+1}{8+9} \times \frac{1+1}{8+9} \\ &= 0.5 \times \frac{2}{17} \times \frac{2}{17} \times \frac{3}{17} \times \frac{2}{17} \end{aligned}$$

Step 5 – Calculate Probabilities (Negative)

Negative:

$$= 0.5 \times \frac{2}{17} \times \frac{0+1}{17} \times \frac{3}{17} \times \frac{2}{17}$$

Note: "love" does not appear in Negative → Laplace smoothing used

Final Calculation (Rough Comparison)

- Positive probability $\approx 0.5 \times \frac{2^3 \cdot 3}{17^4}$
 - Negative probability $\approx 0.5 \times \frac{1 \cdot 2^2 \cdot 3}{17^4}$
- ✓ Positive > Negative → Classify as **Positive**

Language Models

What is a Language Models?

Language models are statistical or neural models that capture the probability distribution of words or tokens in a language.

They predict the likelihood of a sequence of words occurring together, enabling machines to understand and generate human language.

Types of Language Models

1) Statistical Language Models

- **N-gram Models:** Predict the next word based on the previous N-1 words
 - Unigram: $P(\text{word})$ - individual word probabilities
 - Bigram: $P(\text{word}_2 | \text{word}_1)$ - probability of word_2 given word_1
 - Trigram: $P(\text{word}_3 | \text{word}_1, \text{word}_2)$ - probability of word_3 given word_1 and word_2

2) Neural Language Models

- **RNN/LSTM-based:** Process sequences by maintaining hidden states
 - Better at capturing long-range dependencies than n-grams
 - Examples: Early versions of Google Translate
- **Transformer-based:** Use attention mechanisms to process entire sequences
 - BERT: Bidirectional, focuses on understanding context
 - GPT: Autoregressive, focuses on text generation
 - T5, RoBERTa, ELECTRA: Refinements of transformer architecture

Language Models

Trained to predict the next word in a sentence:

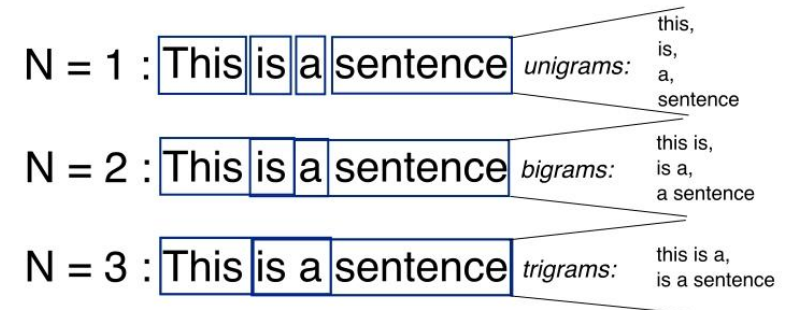
The cat is chasing the _____

dog 5%
mouse 70%
squirrel 20%
boy 5%
house 0%

What are N-grams?

N-grams are a way to break down text into chunks of n words to understand language structure and context better.

- Unigram (1-gram) = single words → "this", "is", "a", "sentence"
- Bigram (2-gram) = pairs of consecutive words → "this is", "is a", "a sentence"
- Trigram (3-gram) = triples of consecutive words → "this is a", "is a sentence"



Implementing Language Models using N-grams

- This practical example will go through the process of creating an N-gram based language model.
- Open the attached Jupyter Notebook to review the example. [Open in Colab](#)

Limitations of N-gram Models

- **Limited context:** Only consider a fixed number of previous words
- **No long-range dependencies:** Cannot capture relationships between distant words
- **No semantic understanding:** Model captures statistical patterns but not meaning
- **Memory intensive:** Storing all possible n-grams requires significant space
- **Data sparsity:** Many valid word combinations don't appear in training data

Perplexity

Perplexity measures how confused a language model is when predicting text.

What It Means

- **Lower perplexity** = Model is more confident and accurate
- **Higher perplexity** = Model is more uncertain and confused

Simple Example

Imagine you're trying to predict the next word in: "I went to the ____"

- A good model might assign high probability to words like "store," "park," "beach"
- A confused model might spread probabilities evenly across thousands of words

Perplexity essentially counts how many reasonable options the model is considering. If the model has a perplexity of 10, it's like the model is equally confused between 10 possible words.

Real-World Comparison

If your phone keyboard suggests 3 very likely next words, it has low perplexity. If it suggests random unrelated words, it has high perplexity.

Why It Matters

Think of perplexity as a "confusion score" that helps researchers:

- Compare different language models
- Check if a model is improving during training
- Identify when a model works better for certain types of text
- The best language models today have much lower perplexity than older models, which is why they produce more natural-sounding text.

How Perplexity is Calculated

Mathematically, for a sequence of words w_1, w_2, \dots, w_N , the perplexity is:

$$\text{Perplexity} = \exp \left(-\frac{1}{N} \sum_{i=1}^N \log P(w_i) \right)$$

This means:

1. You take the log of the probability the model gave for each actual word.
2. You average these.
3. You take the negative exponent to convert back from log space.

Quick Example

Let's say the model is trying to predict a 3-word sentence:

"I love pizza"

And the model assigned these probabilities:

- $P(I) = 0.5$
- $P(\text{love}) = 0.1$
- $P(\text{pizza}) = 0.05$

Then:

$$\text{Perplexity} = \exp \left(-\frac{1}{3}(\log(0.5) + \log(0.1) + \log(0.05)) \right) \approx \exp \left(-\frac{1}{3}(-0.693 - 2.302 - 2.996) \right) \approx \exp(1.997) \approx 7.37$$

So, the model's perplexity here is ~7.37, meaning it's roughly as uncertain as choosing from 7.37 words at each step.