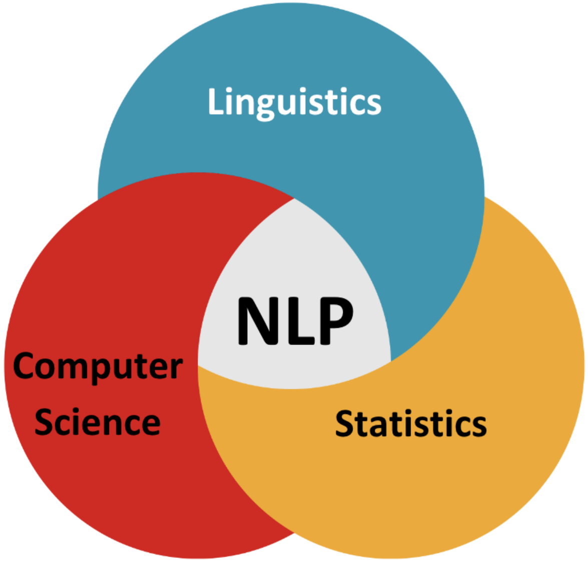
Natural Language Processing

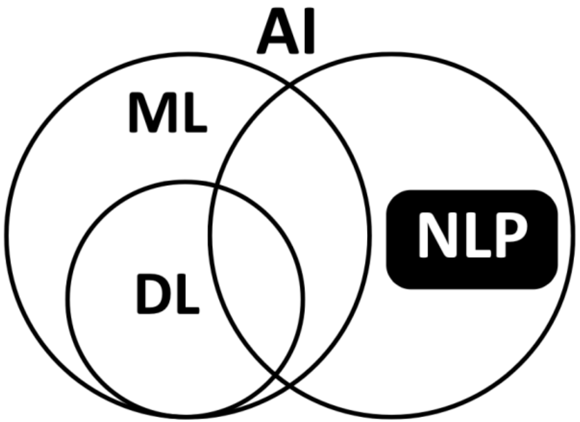
Nicholas Beaudoin | beaudoin@caltech.edu

History of NLP

Introduction to NLP: Definitions and Scope

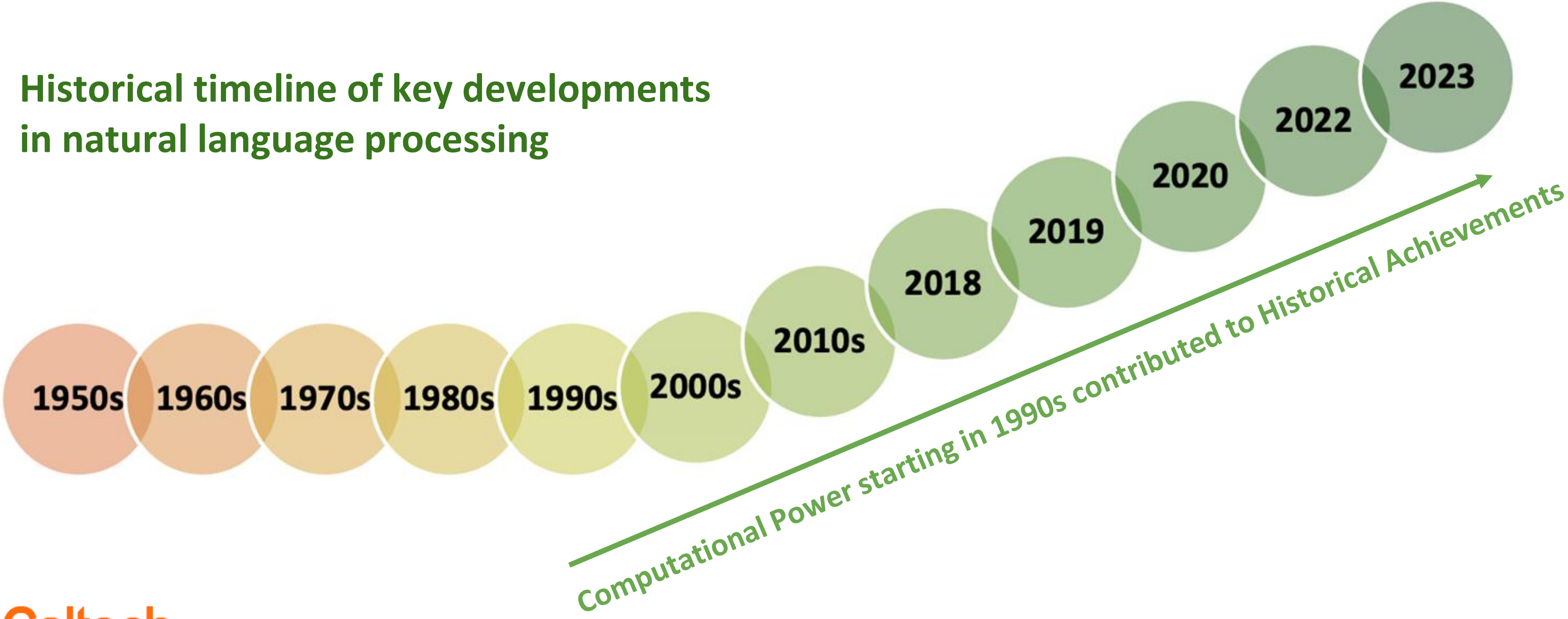


Deriving Meaning from Human Language using Algorithms and Computation



Introduction to NLP: History and Recent Achievements

Historical timeline of key developments in natural language processing



Introduction to NLP: History and Recent Achievements

1950s	Zellig Harris lays the groundwork for Statistical NLP with Distributional Hypothesis
1960s	First machine translations systems
1970s	ELIZA, the first chabot
1980s	Statistical methods gain prominence in NLP such as Hidden Markov Models for Speech Recognition
1990s	Growth of statistical NLP and such as Recurrent Neural Networks for Text Analysis
2000s	Rise of statistical machine translation such as Support Vector Machine for Email Spam Filtering

Introduction to NLP: History and Recent Achievements

2010s	Deep Learning revolution begins in NLP
2018	BERT (Bidirectional Encoder Representations from Transformers) by Google, transforming NLP
2019	GPT-2 released by OpenAI, showing impressive text generation capabilities
2020	GPT-3 released, demonstrating even more advanced language understanding and generation
2022	ChatGPT launched, bringing conversational AI to the mainstream
2023	GPT-4 released, further advancing large language model capabilities

How Did We Get Here?

Fig. Evolution of NLP Models

Source: https://www.freshgravity.com/evolution-of-natural-language-processing/

Text Preprocessing Techniques

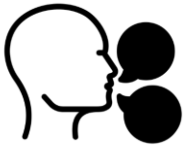
Text Preprocessing - Tokenization

Breaking Text Into Individual Words, Sub-Words, or Characters as Operating Units

Word - Based	Subwords	Character - Based
Learning	Learn ing	L e a r n i n g
Learned	Learn ed	L e a r n e d
Deep Learning	Deep Learn ing	D e e p L e a r n i n g

Vocabulary Size →

Text Preprocessing - Tokenization



Challenges

<u>Contractions</u>

I am	I’m
Is not	Isn’t
He is / He has	He’s
Want to	Wanna

Lookup table for Common Contractions

RegEx Patterns for Complex Cases

Text Preprocessing - Tokenization

Challenges with Identification, Preserving Meaning, other Variations

Compound Words

Closed Compound	**Sunflower, Waterfall, etc.**
Hyphenated Compound	**Long-Term, Mother-in-law, etc.**
Open Compound	**High School, Ice Cream, etc.**

Commonly resolved with Dictionary Based Methods, Pointwise Mutual Information, and ML

Caltech

Text Preprocessing - Tokenization

Workflow Summary

Still Preprocessing and yet to Extract Meaning

Raw Text

Image by Canva

Normalization
Text Cleaning
Compound Words
Special Characters
Machine Learning
Other Processes

Encoding Methods

Embedding Methods

Text Preprocessing - Tokenization

```python
import nltk
nltk.download('punkt')  # Download the Punkt tokenizer data

from nltk.tokenize import word_tokenize

text = "Don't hesitate to ask questions! Mr. Smith's car is parked outside."

tokens = word_tokenize(text)
print(tokens)
```

NLTK

Output

```
['Do', "n't", 'hesitate', 'to', 'ask', 'questions', '!', 'Mr.', 'Smith', "'s",
'car', 'is', 'parked', 'outside', '.']
```

You may need pip install nltk or pip install spacy

Text Preprocessing - Stemming and Lemmatization

```python
from nltk.stem import PorterStemmer

ps = PorterStemmer()

words = ["creates", "creating", "creation", "created", "creative"]

for word in words:
    print(f"{word} -> {ps.stem(word)}")
```

NLTK

create

```
creates -> creat
creating -> creat
creation -> creation
created -> creat
creative -> creativ
```

Stemming only . . . Not always perfect or linguistically accurate

Text Preprocessing - Stemming and Lemmatization

```python
import nltk
nltk.download('wordnet')  # Download WordNet data
from nltk.stem import WordNetLemmatizer

# Initialize the lemmatizer
lemmatizer = WordNetLemmatizer()

# List of words to lemmatize
words = ["create", "creates", "creating", "creation", "created", "creative"]

# Lemmatize each word
for word in words:
    # Lemmatize assuming the word is a verb
    lemma_verb = lemmatizer.lemmatize(word, pos='v')

    # Lemmatize assuming the word is a noun
    lemma_noun = lemmatizer.lemmatize(word, pos='n')

    # Lemmatize assuming the word is an adjective
    lemma_adj = lemmatizer.lemmatize(word, pos='a')

    print(f"Word: {word}")
    print(f"  Lemma (verb): {lemma_verb}")
    print(f"  Lemma (noun): {lemma_noun}")
    print(f"  Lemma (adjective): {lemma_adj}")
    print()
```

```
Word: create
  Lemma (verb): create
  Lemma (noun): create
  Lemma (adjective): create

Word: creates
  Lemma (verb): create
  Lemma (noun): creates
  Lemma (adjective): creates

Word: creating
  Lemma (verb): create
  Lemma (noun): creating
  Lemma (adjective): creating

Word: creation
  Lemma (verb): creation
  Lemma (noun): creation
  Lemma (adjective): creation

Word: created
  Lemma (verb): create
  Lemma (noun): created
  Lemma (adjective): created
```

```
Word: creative
  Lemma (verb): creative
  Lemma (noun): creative
  Lemma (adjective): creative
```

Text Preprocessing - Stemming and Lemmatization

- Stemming reduces words to its root stem meaning, regardless of the context
- Lemmatization working with Stemming reduces words to its accurate root
- Create is the Lemma
- Lemmatization, unlike Stemming, always produces valid words

Image by Canva

Text Preprocessing - Stop Word Removal

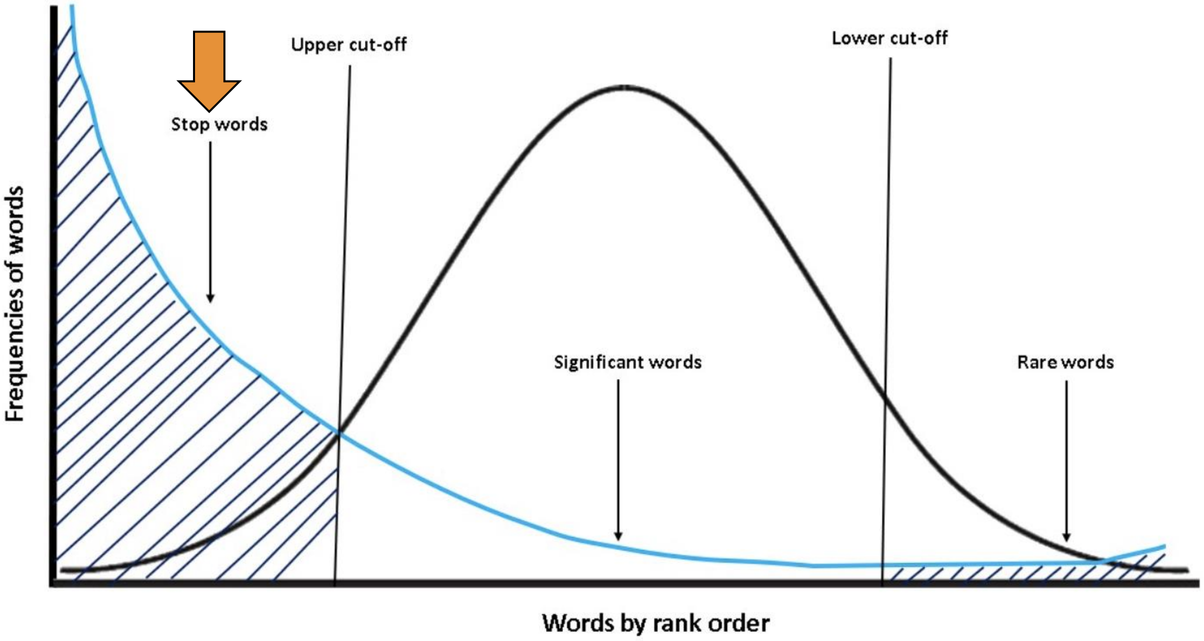


Fig. Graph showing relationship between frequency and rank of words

Image by Wisdom ML
https://wisdomml.in/what-are-stopwords-in-nlp-and-why-we-should-remove-them/

Purpose

- Some words have very little or no meaning at all
- Reduce noise in text data
- Decrease the dimensionality of the feature space
- Focus on more important words that carry more meaning

Text Preprocessing - Stop Word Removal

<u>Common Stop Words</u>

- Articles: "a", "an", "th
- Prepositions: "in", "on", "at", "with", "by
- Pronouns: "I", "he", "she", "it", "we", "they"
- Conjunctions: "and", "but", "or", "nor", "for", "yet", "so"
- Forms of "to be": "am", "is", "are", "was", "were"
- Other common words: "have", "has", "had", "do", "does", "did"

Text Preprocessing - Stop Word Removal

```python
import nltk
nltk.download('punkt')
nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# Sample text
text = "The quick brown fox jumps over the lazy dog. It was a beautiful day."

# Tokenize the text
tokens = word_tokenize(text)

# Get the English stop words
stop_words = set(stopwords.words('english'))

# Remove stop words
filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

print("Original:", tokens)
print("After stop word removal:", filtered_tokens)
```

Original: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog', '.', 'It', 'was', 'a', 'beautiful', 'day', '.']

After stop word removal: ['quick', 'brown', 'fox', 'jumps', 'lazy', 'dog', '.', 'beautiful', 'day', '.']

Text Preprocessing - Handling Punctuation and Special Characters

Importance of Proper Punctuation Handling in NLP

- Strategies for Dealing with Punctuation
  - Removal, Replacement, and Preservation
- Handling Special Characters and Symbols
  - Currency Symbols, brackets, etc.
- Considerations for Domain-Specific Punctuation
  - Social media: hashtags (#)
  - Programming: operators (+, -, *, /), brackets ([]), braces ({})

Image by Canva

Regular Expressions

```python
import re

email = "john.doe@example.com"
pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"

if re.match(pattern, email):
    print("Valid email address")
else:
    print("Invalid email address")
```

Regex Tutorial & Exercise

- Validate a Phone Number
- Extract URLs from Text
- Validate a credit card number
- Extracting Dates from Text
- Validating IP Address
- Extracting Email Addresses
- Exercise

Text Preprocessing - Handling Punctuation and Special Characters

```python
import re
import string

text = "Hello, world! How's it going? #NLP is fun :) Check out https://example.com"

# 1. Remove all punctuation
remove_punct = text.translate(str.maketrans("", "", string.punctuation))
print("Removed punctuation:", remove_punct)

# 2. Replace certain punctuation with tokens
replace_punct = re.sub(r'([.!?])', r' \1 <PUNCT> ', text)
print("Replaced punctuation:", replace_punct)

# 3. Preserve but tokenize around punctuation
preserve_punct = re.findall(r'\w+|[^\w\s]', text)
print("Preserved punctuation:", preserve_punct)

# 4. Handle domain-specific elements (e.g., hashtags, URLs)
handle_special = re.sub(r'(#\w+)', r'<HASHTAG> \1 ', text)
handle_special = re.sub(r'(https?://\S+)', r'<URL> \1 ', handle_special)
print("Handled special elements:", handle_special)
```

- Punctuation can significantly affect meaning and sentiment
- It's crucial for maintaining sentence structure and boundaries

Hands on practice with using text preprocessing [Exercise Placeholder]

Basic Text Analytics

Basic Text Analysis Techniques - Bag of Words

A simple representation of text

- Describes the occurrence of words within a document
- All unique words in the corpus
- Representing each document as a vector of word frequencies

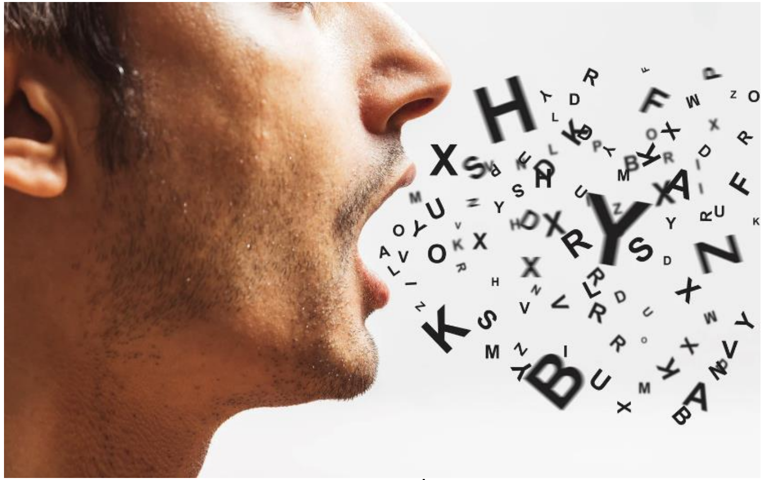


Image by Canva

Basic Text Analysis Techniques - Bag of Words

```python
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from collections import Counter

# Download necessary NLTK data
nltk.download('punkt')
nltk.download('stopwords')

def preprocess(text):
    # Tokenize the text
    tokens = word_tokenize(text.lower())

    # Remove stopwords and non-alphabetic tokens
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token.isalpha() and token not in stop_words]

    return tokens

def create_bow(documents):
    # Create vocabulary
    vocab = set()
    for doc in documents:
        vocab.update(preprocess(doc))

    # Create BoW representation for each document
    bow_representations = []
    for doc in documents:
        bow = Counter(preprocess(doc))
        bow_vector = [bow.get(word, 0) for word in vocab]
        bow_representations.append(bow_vector)

    return list(vocab), bow_representations
```

The word bag is used because there is no order in the words, which may lose context

Code Demonstration with BagOfWords.py

```python
# Example usage
documents = [
    "The cat sat on the mat.",
    "The dog chased the cat.",
    "The mat was on the floor."
]

vocabulary, bow_representations = create_bow(documents)

print("Vocabulary:")
print(vocabulary)
print("\nBag of Words representations:")
for i, bow in enumerate(bow_representations):
    print(f"Document {i + 1}: {bow}")
```

Basic Text Analysis Techniques - N grams

Words that are next to each other in a sequence is a <u>gram</u> in NLP

- Unigrams: ["The", "cat", "sat", "on", "the", "mat"]

  n = 1
- Bigrams: ["The cat", "cat sat", "sat on", "on the", "the mat"] n = 2
- Trigrams: ["The cat sat", "cat sat on", "sat on the", "on the mat"] n = 3

Basic Text Analysis Techniques - N grams

```python
import nltk
from nltk import word_tokenize, ngrams
from collections import Counter

# Download necessary NLTK data
nltk.download('punkt')

def generate_ngrams(text, n):
    # Tokenize the text
    tokens = word_tokenize(text.lower())

    # Generate n-grams
    n_grams = list(ngrams(tokens, n))

    return n_grams

def analyze_ngrams(text):
    print(f"Original text: {text}")

    # Generate and analyze unigrams, bigrams, and trigrams
    for n in range(1, 4):
        n_grams = generate_ngrams(text, n)

        print(f"\n{n}-grams:")
        for gram in n_grams:
            print(f"  {' '.join(gram)}")

        # Count frequency of each n-gram
        gram_freq = Counter(n_grams)

        print(f"\nMost common {n}-grams:")
        for gram, count in gram_freq.most_common(3):
            print(f"  {' '.join(gram)}: {count}")

# Example usage
text = "The quick brown fox jumps over the lazy dog. The dog was not amused."

analyze_ngrams(text)
```

Capture some context and word order after Bag of Words

Code demonstration of nGrams.py

```python
# Bonus: Using n-grams for text generation
def generate_text(text, n, num_words):
    tokens = word_tokenize(text.lower())
    n_grams = list(ngrams(tokens, n))

    # Start with a random n-gram
    import random
    current = random.choice(n_grams)
    result = list(current)

    for _ in range(num_words - n):
        possible_next = [gram for gram in n_grams if gram[:-1] == current[1:]]
        if not possible_next:
            break
        next_gram = random.choice(possible_next)
        result.append(next_gram[-1])
        current = next_gram

    return ' '.join(result)

# Generate text using trigrams
generated_text = generate_text(text, 3, 20)
print("\nGenerated text using trigrams:")
print(generated_text)
```

Basic Text Analysis Techniques - Term frequency-inverse document frequency (TF-IDF)

TF - IDF is a Numerical Statistic that reflect how important a word is to documents

- **Term Frequency (TF)** measures how frequently a term occurs in a document
- **Inverse Document Frequency (IDF)** measures how important a term is across the entire corpus
- **TF - IDF = TF times IDF**

Weighing Words in Text Analytics

Basic Text Analysis Techniques -
Term frequency-inverse document frequency (TF-IDF)

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

# Sample documents
documents = [
    "The cat sat on the mat",
    "The dog chased the cat",
    "The mat was on the floor"
]

# Create TfidfVectorizer object
vectorizer = TfidfVectorizer()

# Calculate TF-IDF
tfidf_matrix = vectorizer.fit_transform(documents)

# Get feature names (words)
feature_names = vectorizer.get_feature_names_out()

# Convert to DataFrame for better visualization
df = pd.DataFrame(
    tfidf_matrix.toarray(),
    columns=feature_names,
    index=['Doc 1', 'Doc 2', 'Doc 3']
)

print("TF-IDF Matrix:")
print(df)
```

Code Demonstration with TF-IDF.py

```
# Calculate and print IDF values
idf_values = vectorizer.idf_
idf_df = pd.DataFrame(
    {'Term': feature_names, 'IDF': idf_values}
).sort_values(by='IDF', ascending=False)

print("\nIDF Values:")
print(idf_df)

# Function to explain TF-IDF calculation for a specific term in a document
def explain_tfidf(term, doc_index):
    term_index = list(feature_names).index(term)
    tf = vectorizer.transform([documents[doc_index]]).toarray()[0][term_index]
    idf = vectorizer.idf_[term_index]
    tfidf = tf * idf

    print(f"\nExplanation for term '{term}' in Document {doc_index + 1}:")
    print(f"TF (Term Frequency): {tf}")
    print(f"IDF (Inverse Document Frequency): {idf:.4f}")
    print(f"TF-IDF: {tfidf:.4f}")

# Example explanation
explain_tfidf("cat", 0)  # Explain 'cat' in the first document
```

Frequency adjustment of words in text data

Basic Word cloud representations

Reference: https://towardsdatascience.com/create-word-cloud-into-any-shape-you-want-using-python-d0b88834bc32

Advanced NLP Techniques: Sentiment Analysis

We need to understand how our customers feel about our product

Traditionally, we can let our customers use a 1-5 rating system to get feedback

Source: yelp.com/biz/bird-rock-coffee-roasters-waterfront-san-diego

We need to understand how our customers feel about our product

What if we want to look deeper at the language used or if we don't have a rating system

An amazing find!

Nestled in the lobby of an office building, it looked like it was going to be another bland corporate coffee shop.

It was surprising at how laid back and comfortable the lounge was.

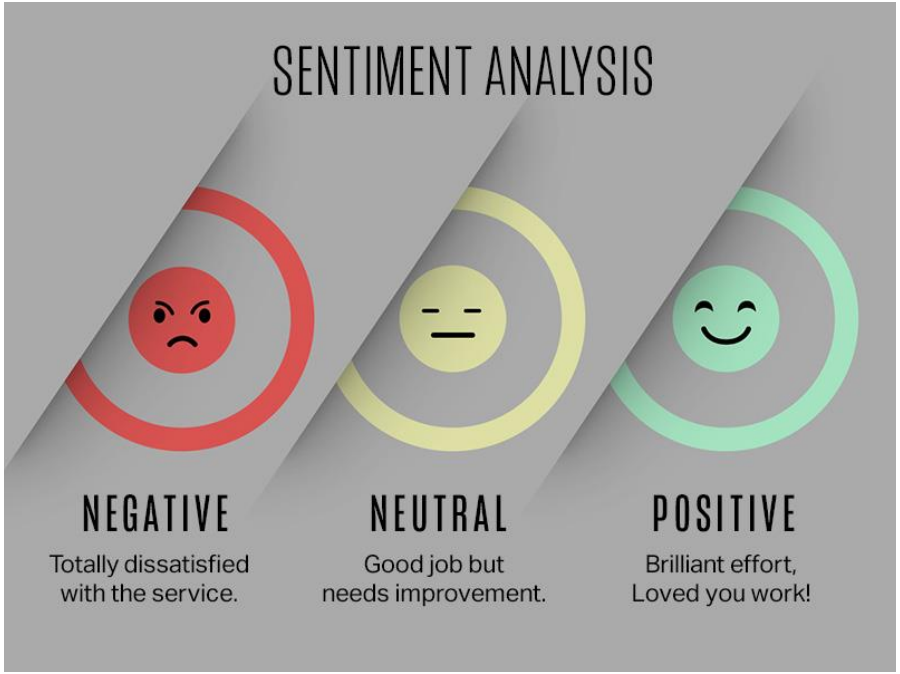
The service and coffee were also excellent. The staff was efficient and helpful.

This is now on my list of standby coffee shops in San Diego.

Select your rating

Source: yelp.com/biz/bird-rock-coffee-roasters-waterfront-san-diego

Sentiment Analysis takes raw text data into customer experience insights



Source: https://x.com/YashimVG/status/1756972282881155307

Two methods: Lexicon-Based Sentiment Analysis

What is it?

A rule-based approach to sentiment analysis that relies on predefined dictionaries (lexicons) of words associated with sentiment values.

How It Works:

1. **Lexicon/Dictionaries**:
   - Words are labeled as positive, negative, or neutral (e.g., "good" = +1, "bad" = -1).
2. **Scoring**:
   - Each word in the text is matched against the lexicon.
   - Sentiment score = sum of sentiment values of matched words.
3. **Interpretation**:
   - Positive score → Positive sentiment.
   - Negative score → Negative sentiment.

Technical Components of Lexicon-based Sentiment Analysis

```python
# A simple example of a lexicon (dictionary of sentiment scores)
lexicon = {
    "great": 2,
    "excellent": 3,
    "bad": -2,
    "terrible": -3,
    "not": -1  # negation handling
}

# Text preprocessing (removing punctuation, lowercasing, tokenizing)
def preprocess(text):
    text = text.lower()
    text = re.sub(r'[^\w\s]', '', text)
    return text.split()

# Calculating sentiment score
def calculate_sentiment(text, lexicon):
    tokens = preprocess(text)
    sentiment_score = 0
    negation_flag = False

    for token in tokens:
        if token in lexicon:
            score = lexicon[token]
            # Handle negation
            if negation_flag:
                score = -score
                negation_flag = False  # Reset after applying negation
            sentiment_score += score
        if token == "not":  # Set flag for negation
            negation_flag = True
```

```python
    # Determine sentiment category
    if sentiment_score > 0:
        sentiment = "Positive"
    elif sentiment_score < 0:
        sentiment = "Negative"
    else:
        sentiment = "Neutral"

    return sentiment, sentiment_score

# Example usage
text = "The service was not bad, but the food was excellent."
sentiment, score = calculate_sentiment(text, lexicon)
print(f"Text: '{text}'")
print(f"Sentiment: {sentiment} (Score: {score})")
```

Two Methods: Machine-Learning-Based Methods

What is it?

Machine learning approaches rely on labeled training data to teach a model how to classify sentiment. These methods are dynamic and can adapt to new data, making them highly effective for complex or large-scale tasks.

Strengths:

- **Dynamic and Adaptive**: Learns from domain-specific language.
- **Contextual Understanding**: Handles combinations of words (e.g., "not bad") better than lexicon-based approaches.
- **Scalable**: Works well with large datasets.

Weaknesses:

- **Requires Labeled Data**: Model training depends on having sufficient high-quality labeled examples.
- **Computationally Intensive**: Preprocessing/training/optimization require more resources.
- **Overfitting**: Without proper regularization or validation, the model might perform well on training data but poorly on unseen data.

Technical Components of ML Based Sentiment Analysis

How It Works:

1. **Training Phase**:
    - **Data Collection**: Labeled examples (positive/negative reviews).
    - **Preprocessing**: Cleaning + transforming text into numerical representations (like TF-IDF or embeddings).
    - **Model Training**: Use models like Logistic Regression or Neural Networks learn patterns in the training data.
2. **Prediction Phase**:
    - New, unseen text is preprocessed in the same way.
    - The trained model predicts the sentiment label (positive, negative, or neutral).

Social Listening powered by Sentiment Analysis

Source: https://www.diligent.es/social-media-listening-strategy-starbucks/

Sentiment Analysis Lab

Advanced NLP Techniques: Topic Modeling

What is Topic Modeling?

- ML Technique used in Natural Language Processing (NLP) to identify underlying topics or themes in a large corpus of text data.
- The goal of topic modeling is to automatically discover and extract meaningful topics or themes from a large collection of text documents,
- No prior knowledge or labeling of the data is necessary

Topic modeling: corpus of text data -> mixture of topics,

Topic -> Distribution over a set of words.

Goal: Learn the underlying topics and their corresponding word distributions.

Technical Details of Topic Modeling

Topic modeling has many applications in NLP, including:

1. Text classification: Topic modeling can be used to improve text classification by identifying the most relevant topics in the text data.
2. Information retrieval: Topic modeling can be used to improve information retrieval by identifying the most relevant topics in a search query.
3. Sentiment analysis: Topic modeling can be used to identify the sentiment of text data by identifying the topics that are associated with positive or negative sentiment.
4. Document summarization: Topic modeling can be used to summarize long documents by identifying the most important topics and sentences.

Use case: Why would we use topic modeling?

1. Improved text understanding: Topic modeling can help improve text understanding by identifying the underlying topics and themes in the text data.

2. Reduced dimensionality: Topic modeling can reduce the dimensionality of the text data by identifying the most important topics and ignoring the rest.
3. Improved clustering: Topic modeling can improve clustering by identifying the most relevant topics and grouping similar documents together.

Topic Modeling Lab

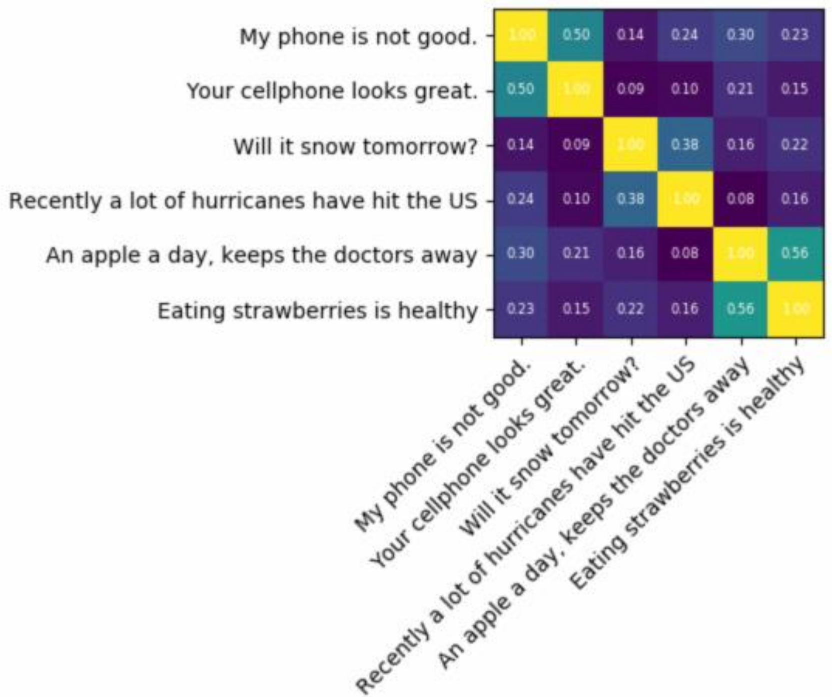
Advanced NLP Techniques: Document Similarity

What is Document Similarity?

Document similarity is process of quantifying how closely two or more pieces of text are related in terms of content, structure, or meaning.

We use it to:

- Identify relationships between texts.
- Measure overlap in information or themes.
- Group similar documents or retrieve relevant ones.



Source: https://stackoverflow.com/questions/8897593/how-to-compute-the-similarity-between-two-text-documents

Lexical vs. Semantic Similarity

Lexical Similarity:

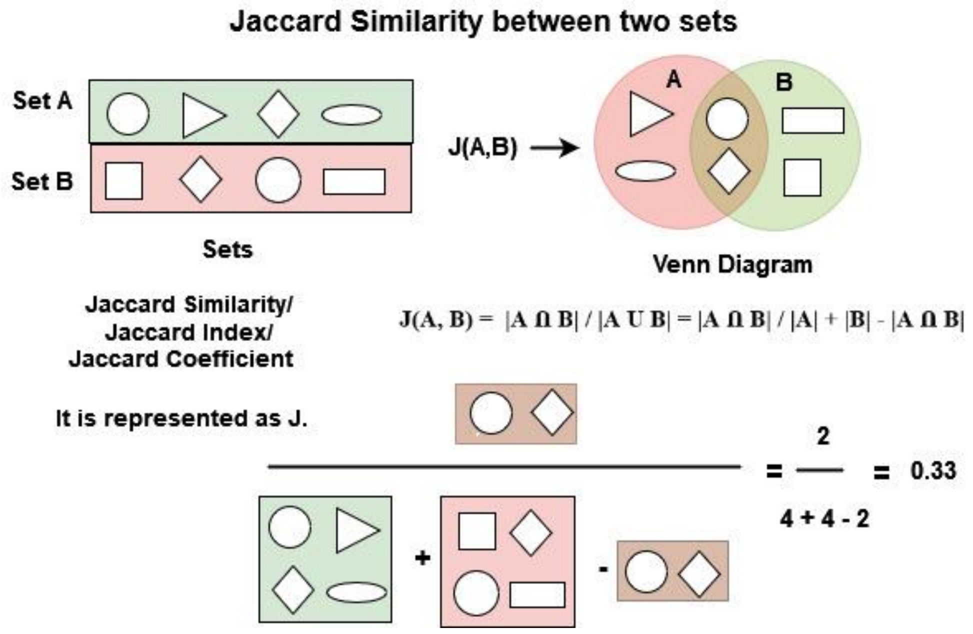
- Measures similarity based on **word overlap**.
- Techniques:
  - Jaccard Similarity (set-based overlap).
  - Cosine Similarity with Bag-of-Words or TF-IDF.
- Pros:
  - Simple, interpretable, computationally efficient.
  - Works well for direct word matches.
- Cons:
  - Misses **synonyms** and cannot handle **context**.
  - Example: *"doctor"* and *"physician"* → Low similarity due to no word overlap.

Semantic Similarity:

- Measures similarity based on **meaning** and **context**.
- Techniques:
  - Word Embeddings (Word2Vec, GloVe).
  - Transformer models (BERT).
- Pros:
  - Captures **synonyms**, related words, and deeper relationships.
  - Effective for nuanced text comparisons: *"AI"* and *"Artificial Intelligence"* → High similarity despite no word overlap.
- Cons:
  - Computationally intensive, requires **pre-trained models**.

Jaccard Similarity

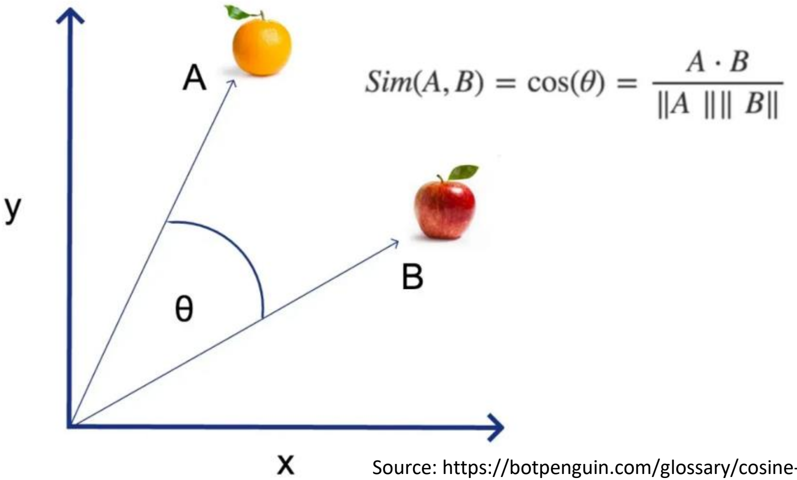
- Measures the similarity between two sets by calculating the ratio of their intersection to their union.
- Example:
  - Text 1: *"data science is amazing"*
  - Text 2: *"science of data is great"*
  - Common words: {"data", "science", "is"} → High similarity.
- Ignores word order and frequency.
- Misses semantic relationships (e.g., synonyms).



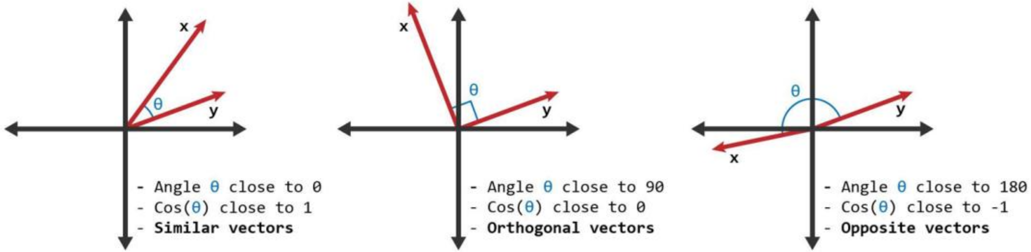
Source: https://medium.com/@rohan10dalvi/jaccard-similarity-in-graph-theory-a049d6ba8a9

Cosine Similarity

- Measures the similarity between two vectors by calculating the cosine of the angle between them.
- Handles word frequency and importance (unlike Jaccard)
- Works well for high-dimensional, sparse data.
- Ignores semantic meaning



Source: https://botpenguin.com/glossary/cosine-similarity



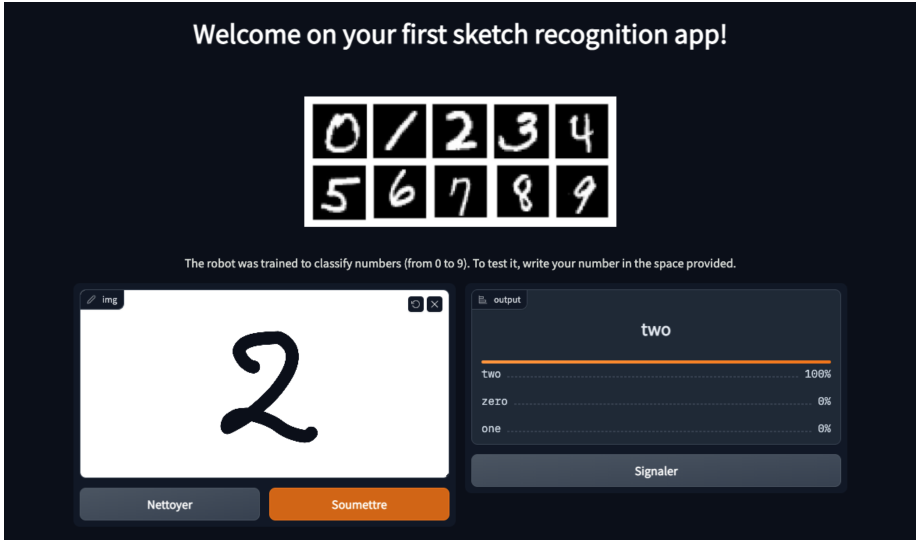
Source: https://www.learndatasci.com/glossary/cosine-similarity/

Document Similarity Lab

Introduction to Gradio

What is Gradio?

- Gradio is a Python library that allows you to quickly build **interactive web interfaces** for machine learning models, APIs, or data pipelines.
- Enables users to interact with models through a simple GUI without writing extensive front-end code.
- Supports text, images, audio, video, sliders, etc. as input.



Source: https://help.ovhcloud.com/csm/es-public-cloud-ai-deploy-gradio-sketch-recognition?id=kb_article_view&sysparm_article=KB0048098

Gradio Basics Lab

Building our own End-to-End NLP Solution

End-to-end NLP Solution

1. Read Data and preprocess it
2. Visualize it in a word cloud/frequency distribution
3. Apply two techniques
   a. Sentiment Analysis
   b. Topic Modelling

How does your solution impact businesses?

Use case: Contract Research

Source: https://www.axiomlaw.com/guides/types-of-contracts