Tanmay Suhas Jagtap

Portfolio : https://78-t0b1.github.io/Portfolio.github.io/

Linkein : https://www.linkedin.com/in/tanmay-jagtap-t0b1/

Github : https://github.com/78-t0b1

Week 1

What is NLP?

IBM

Natural language processing (NLP) is a subfield of computer science and artificial intelligence (AI) that uses machine learning to enable computers to understand and communicate with human language.

What are its applications?

Finance: In financial dealings, nanoseconds might make the difference between success and failure when accessing data, or making trades or deals. NLP can speed the mining of information from financial statements, annual and regulatory reports, news releases or even social media.

Healthcare: New medical insights and breakthroughs can arrive faster than many healthcare professionals can keep up. NLP and AI-based tools can help speed the analysis of health records and medical research papers, making better-informed medical decisions possible, or assisting in the detection or even prevention of medical conditions.

Insurance: NLP can analyze claims to look for patterns that can identify areas of concern and find inefficiencies in claims processing—leading to greater optimization of processing and employee efforts.

Legal: Almost any legal case might require reviewing mounds of paperwork, background information and legal precedent. NLP can help automate legal discovery, assisting in the organization of information, speeding review and helping ensure that all relevant details are captured for consideration.

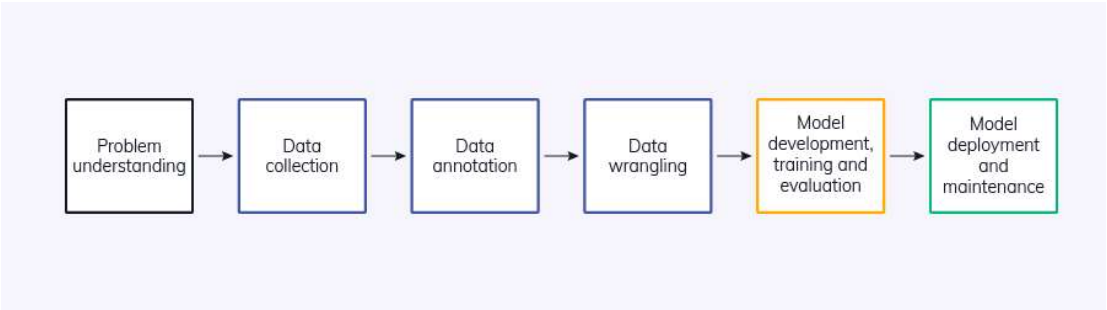
Types of NLP business applications : twilio blog

- Text to speech: Converting text-to-speech data, then reproducing the text as natural-sounding speech
- Chatbots: Helping chatbots understand and respond to customer inquiries
- Urgency detection: Analyzing language to prioritize tasks

- Natural language understanding: Converting speech to text and analyzing its intent
- Autocorrect: Detecting and removing text errors and suggesting alternatives
- Sentiment analysis: Revealing the perceptions people have of your goods and services and those of your competitors
- Speech recognition: Powering applications that understand users' voices and deciphering their meaning

Lifecycle of NLP project

Neptune AI blog



1. Problem Understanding

- Clearly define the objectives and requirements of the NLP project, such as whether it will be integrated into a chatbot, used for a specific industry, or designed as a generic model.
- Understand the context and input types the NLP model needs to handle, such as emojis, titles, and conversational context.
- Clarify the purpose and intended use case of the sentiment analysis or other NLP capabilities.

2. Data Collection and Labeling

- Determine if the client has provided any initial data, or if you need to start from scratch with data collection.
- Explore techniques to handle limited data, such as using GPT models for synthetic data generation or leveraging data labeling tools like Argilla.
- Invest significant time and resources into high-quality data collection and annotation, as this is crucial for building an effective NLP model.

3. Data Preparation and Preprocessing

- Perform lexical analysis to segment text into meaningful units like words, phrases, and sentences.
- Conduct syntactic analysis to check the grammatical structure of the text.
- Carry out semantic analysis to understand the meaning and context of the text.
- Handle other preprocessing steps like stop word removal, stemming/lemmatization, and feature engineering.

4. Model Development and Training

- Select appropriate NLP models and techniques, such as traditional machine learning algorithms or newer transformer-based models like BERT.
- Train the NLP models using the prepared dataset, optimizing hyperparameters and monitoring performance.
- Evaluate the model's performance using relevant metrics like accuracy, precision, recall, and F1-score.

5. Model Deployment and Maintenance

- Integrate the trained NLP model into the production environment, whether that's a chatbot, an analytics system, or some other application.
- Monitor the model's performance in the real-world setting and fine-tune or retrain it as needed to maintain high accuracy.
- Ensure the NLP system can handle evolving language patterns, new data sources, and changing user requirements over time.

Throughout this lifecycle, it's important to maintain close collaboration between domain experts, data scientists, and software engineers to ensure the NLP project meets the desired business objectives. The iterative nature of NLP projects also requires flexibility and a willingness to revisit earlier stages as needed to improve the model's performance and robustness.

Problem Understanding

Understanding the different algorithms and their applications in Natural Language Processing (NLP) is crucial for solving various text-based problems. Here's a detailed breakdown of common NLP tasks, the algorithms used for each task, and their applications:

1. Sentiment Analysis

Purpose: Determine the sentiment (positive, negative, neutral) expressed in text.

Common Algorithms:

- **Lexicon-Based Methods**: Use predefined dictionaries of words associated with positive or negative sentiments. Examples include VADER (Valence Aware Dictionary and sEntiment Reasoner) and SentiWordNet.
    - **When to Use**: Quick and interpretable sentiment analysis on relatively simple texts where context and sarcasm are not major concerns.
- **Machine Learning Classifiers**: Use algorithms like Naive Bayes, Support Vector Machines (SVM), or Logistic Regression trained on labeled datasets.
    - **When to Use**: When you have labeled training data and need more flexibility than lexicon-based methods.
- **Deep Learning Models**: Use Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and transformer-based models like BERT for more

accurate sentiment analysis.

- **When to Use**: When dealing with complex texts where context, syntax, and semantics play a significant role.

2. Named Entity Recognition (NER)

Purpose: Identify and classify named entities (people, organizations, locations, etc.) in text.

Common Algorithms:

- **Conditional Random Fields (CRFs)**: Probabilistic models used for structured prediction.
  - **When to Use**: Traditional approach, effective for many NER tasks with structured and labeled training data.
- **Neural Networks**: Use architectures like BiLSTM-CRF to combine neural networks with CRFs for improved performance.
  - **When to Use**: When you need more sophisticated models that can capture complex dependencies in the text.
- **Transformer Models**: Models like BERT, RoBERTa, and GPT-3 fine-tuned for NER tasks.
  - **When to Use**: When state-of-the-art performance is required, particularly for texts with rich contextual information.

3. Text Classification

Purpose: Assign predefined categories or labels to text.

Common Algorithms:

- **Naive Bayes**: Probabilistic classifier based on Bayes' theorem.
  - **When to Use**: Simple, fast, and effective for many text classification tasks, especially with small datasets.
- **Support Vector Machines (SVM)**: Supervised learning models that can perform linear and non-linear classification.
  - **When to Use**: When you need robust performance on medium-sized datasets and can afford the computational cost.
- **Convolutional Neural Networks (CNNs)**: Capture local patterns in text, often used for short text classification.
  - **When to Use**: When classifying text that benefits from capturing spatial hierarchies (e.g., sentiment in sentences).
- **Transformers**: BERT, DistilBERT, and other transformer models fine-tuned for text classification.

    - **When to Use**: For high performance on complex classification tasks with sufficient computational resources.

4. Machine Translation

Purpose: Translate text from one language to another.

Common Algorithms:

- **Statistical Machine Translation (SMT)**: Uses statistical models based on the analysis of bilingual text corpora.
    - **When to Use**: Older approach, less common now, suitable when limited computational resources are available.
- **Neural Machine Translation (NMT)**: Uses neural networks to model the entire translation process.
    - **When to Use**: When you have sufficient data and computational power to train neural models.
- **Transformer-Based Models**: Use models like Transformer, BERT, and GPT for translation tasks.
    - **When to Use**: For state-of-the-art translation performance, particularly in complex and high-resource language pairs.

5. Summarization

Purpose: Generate concise summaries of longer text.

Common Algorithms:

- **Extractive Summarization**: Selects key sentences from the original text to form the summary. Algorithms include TextRank and other graph-based methods.
    - **When to Use**: When you need quick, interpretable summaries that preserve the original text's meaning.
- **Abstractive Summarization**: Generates summaries by creating new sentences, using models like seq2seq (sequence-to-sequence) with attention mechanisms, and T5 (Text-to-Text Transfer Transformer).
    - **When to Use**: When you need more natural and human-like summaries that may rephrase or paraphrase the original text.

Data Preparation and Preprocessing

In NLP data preperation in diffrent than other datsets. Main reson is data we are handeling here is unstructured. Text pre-processing is the process of transforming unstructured text to structured text to prepare it for analysis. To understand text pre processing first lets understand what does word means.

Word

Speech and Language Processing

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

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

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

Should we consider a capitalized string (like They) and one that is uncapitalized (like they) to be the same word type? The answer is that it depends on the task! They and they might be lumped together as the same type in some tasks, like speech recognition, where we might just care about getting the words in order and don't care about the formatting, while for other tasks, such as deciding whether a particular word is a noun or verb (part-of-speech tagging) or whether a word is a name of a person or location (named-entity tagging), capitalization is a useful feature and is retained. Sometimes we keep around two versions of a particular NLP model, one with capitalization and one without capitalization.

Because language is so situated, when developing computational models for language processing from a corpus, it's important to consider who produced the language, in what context, for what purpose. How can a user of a dataset know all these details? The best way is for the corpus creator to build a datasheet (Gebru et al., 2020) or data statement (Bender et al., 2021) for each corpus. A datasheet specifies properties of a dataset like:

Motivation: Why was the corpus collected, by whom, and who funded it?

Situation: When and in what situation was the text written/spoken? For example, was there a task? Was the language originally spoken conversation, edited text, social media communication, monologue vs. dialogue?

Language variety: What language (including dialect/region) was the corpus in?

Speaker demographics: What was, e.g., the age or gender of the text's authors?

Collection process: How big is the data? If it is a subsample how was it sampled? Was the data collected with consent? How was the data pre-processed, and what metadata is available?

Annotation process: What are the annotations, what are the demographics of the annotators, how were they trained, how was the data annotated?

Distribution: Are there copyright or other intellectual property restrictions?

Text Normalization

At least three tasks are commonly applied as part of any normalization process:

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

1. Tokenization

"Tokenization is the process of breaking down a piece of text, like a sentence or a paragraph, into individual words or "tokens." These tokens are the basic building blocks of language, and tokenization helps computers understand and process human language by splitting it into manageable units."

There are roughly two classes of tokenization algorithms.

In top-down tokenization, we define a standard and implement rules to implement that kind of tokenization.

In bottom-up tokenization, we use simple statistics of letter sequences to break up words into subword tokens.

Top down Tokenization:

Top-down (rule-based) tokenization is a method for dividing text into smaller components (tokens) using predefined rules. This approach uses a set of deterministic rules to identify boundaries between tokens, such as words, punctuation marks, and other meaningful elements.

Key Features of Top-Down (Rule-Based) Tokenization:

1. Deterministic: The rules are fixed and do not change, leading to predictable results.
2. Predefined Rules: These rules are usually handcrafted by linguists or experts in the language.
3. Simplicity: Often simpler and faster than statistical methods since they do not rely on training data.

Example: "Email me at user@example.com" → ["Email", "me", "at", "user", "@", "example", ".", "com"]

In [ ]:
```
import re

def simple_tokenizer(text):
    # Define the tokenization pattern
    pattern = r"[\w']+|[.,!?;]"
    return re.findall(pattern, text)

# Example usage
text = "Hello, world! It's a well-known fact."
```

```
tokens = simple_tokenizer(text)
print(tokens)
```

```
['Hello', ',', 'world', '!', "It's", 'a', 'well', 'known', 'fact', '.']
```

Bottom-up Tokenization:

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

To deal with this unknown word problem, modern tokenizers automatically insubwords duce sets of tokens that include tokens smaller than words, called subwords. Subwords can be arbitrary substrings, or they can be meaning-bearing units like the morphemes -est or -er. (A morpheme is the smallest meaning-bearing unit of a language; for example the word unlikeliest has the morphemes un-, likely, and -est.)

In modern tokenization schemes, most tokens are words, but some tokens are frequently occurring morphemes or other subwords like -er. Every unseen word like lower can thus be represented by some sequence of known subword units, such as low and er, or even as a sequence of individual letters if necessary.

Byte-pair Encoding

BPE starts with the smallest possible units (individual characters) and iteratively merges the most frequent pairs of symbols to form larger units. This process continues until a predefined vocabulary size is reached.

Steps of Byte Pair Encoding

1. Initialize Vocabulary: Start with a vocabulary that includes all individual characters in the text corpus.
2. Count Pairs: Count all adjacent character pairs in the corpus.
3. Merge the Most Frequent Pair: Find the most frequent pair of characters and merge them into a new symbol.
4. Repeat: Repeat the process of counting and merging until the desired vocabulary size is achieved.

Byte Pair Encoding is a powerful tokenization method widely used in NLP, especially in models like GPT and BERT. It balances the need for a manageable vocabulary size with the ability to handle rare and out-of-vocabulary words effectively.

Implimentation:

Corpus: low low lower newer

In [ ]:
```
from collections import defaultdict, Counter

def get_stats(vocab):
    pairs = defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols) - 1):
            pairs[symbols[i], symbols[i + 1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    v_out = {}
    bigram = ' '.join(pair)
    replacement = ''.join(pair)
    for word in v_in:
        w_out = word.replace(bigram, replacement)
        v_out[w_out] = v_in[word]
    return v_out

# Example vocabulary
vocab = {
    'l o w': 2,
    'l o w e r': 1,
    'n e w e r': 1,
    'l o w e s t': 1
}

num_merges = 10
for i in range(num_merges):
    pairs = get_stats(vocab)
    if not pairs:
        break
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
    print(f'Merge {i+1}: {best}')
    print(vocab)

print(vocab)
```

```
Merge 1: ('l', 'o')
{'lo w': 2, 'lo w e r': 1, 'n e w e r': 1, 'lo w e s t': 1}
Merge 2: ('lo', 'w')
{'low': 2, 'low e r': 1, 'n e w e r': 1, 'low e s t': 1}
Merge 3: ('low', 'e')
{'low': 2, 'lowe r': 1, 'n e w e r': 1, 'lowe s t': 1}
Merge 4: ('lowe', 'r')
{'low': 2, 'lower': 1, 'n e w e r': 1, 'lowe s t': 1}
Merge 5: ('n', 'e')
{'low': 2, 'lower': 1, 'ne w e r': 1, 'lowe s t': 1}
Merge 6: ('ne', 'w')
{'low': 2, 'lower': 1, 'new e r': 1, 'lowe s t': 1}
Merge 7: ('new', 'e')
{'low': 2, 'lower': 1, 'newe r': 1, 'lowe s t': 1}
Merge 8: ('newe', 'r')
{'low': 2, 'lower': 1, 'newer': 1, 'lowe s t': 1}
Merge 9: ('lowe', 's')
{'low': 2, 'lower': 1, 'newer': 1, 'lowes t': 1}
Merge 10: ('lowes', 't')
{'low': 2, 'lower': 1, 'newer': 1, 'lowest': 1}
{'low': 2, 'lower': 1, 'newer': 1, 'lowest': 1}
```

2. Word Normalization, Lemmatization and Stemming

"Word normalization is the task of putting words/tokens in a standard format."

The case folding simplest case of word normalization is case folding. Mapping everything to lower case means that Woodchuck and woodchuck are represented identically, which is very helpful for generalization in many tasks, such as information retrieval or speech recognition.

Systems that use BPE or other kinds of bottom-up tokenization may do no further word normalization. In other NLP systems, we may want to do further normalizations, like choosing a single normal form for words with multiple forms like USA and US or uh-huh and uhhuh.

Lemmatization

"Lemmatization is the task of determining that two words have the same root, despite their surface differences."

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

In [ ]:
```python
import spacy

# Load the spaCy model
nlp = spacy.load("en_core_web_sm")

# Example text
text = "The children were running and had eaten their meals."

# Process the text
```

```
doc = nlp(text)

# Print tokens and their Lemmas
for token in doc:
    print(f"Token: {token.text}         Lemma: {token.lemma_}")
```

```
Token: The         Lemma: the
Token: children         Lemma: child
Token: were         Lemma: be
Token: running         Lemma: run
Token: and         Lemma: and
Token: had         Lemma: have
Token: eaten         Lemma: eat
Token: their         Lemma: their
Token: meals         Lemma: meal
Token: .         Lemma: .
```

Stemming

Lemmatization algorithms can be complex. For this reason we sometimes make use of a simpler but cruder method, which mainly consists of chopping off word final affixes. This naive version of morphological analysis is called stemming.

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

produces the following stemmed output:

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

In [ ]:
```
import nltk
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

# Download the required NLTK data
nltk.download('punkt')

# Example text
text = "The children were running and had eaten their meals."

# Tokenize the text
tokens = word_tokenize(text)

# Initialize the Porter Stemmer
porter = PorterStemmer()

# Apply stemming
stems = [porter.stem(token) for token in tokens]

# Print tokens and their stems
for token, stem in zip(tokens, stems):
    print(f"Token: {token}       Stem: {stem}")
```

```
Token: The       Stem: the
Token: children       Stem: children
Token: were       Stem: were
Token: running       Stem: run
Token: and       Stem: and
Token: had       Stem: had
Token: eaten       Stem: eaten
Token: their       Stem: their
Token: meals       Stem: meal
Token: .       Stem: .
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\tanma\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

3. Sentence Segmentation

Sentence segmentation is another important step in text processing. The most useful cues for segmenting a text into sentences are punctuation, like periods, question marks, and exclamation points. In general, sentence tokenization methods work by first deciding (based on rules or machine learning) whether a period is part of the word or is a sentence-boundary marker. An abbreviation dictionary can help determine whether the period is part of a commonly used abbreviation; the dictionaries can be hand-built or machinelearned (Kiss and Strunk, 2006), as can the final sentence splitter.

In [ ]:
```python
from nltk.tokenize import sent_tokenize

# Example text
text = "Hello! How are you? I hope you're doing well. Have a great day!"

# Segment the text into sentences
sentences = sent_tokenize(text)

# Print each sentence
for sentence in sentences:
    print(sentence)
```

```
Hello!
How are you?
I hope you're doing well.
Have a great day!
```

Stopwords removal

Stop word removal is a common preprocessing step in natural language processing (NLP) where commonly occurring words (e.g., "the", "is", "in") are removed from a text. These words are usually filtered out because they do not carry significant meaning and can reduce the efficiency of text processing tasks.

In [ ]:
```python
import spacy

# Load the spaCy model
nlp = spacy.load("en_core_web_sm")

# Example text
text = "This is a simple example to demonstrate stop word removal."
```

```
# Process the text
doc = nlp(text)

# Remove stop words from the tokens
filtered_tokens = [token.text for token in doc if not token.is_stop]

# Print the filtered tokens
print(filtered_tokens)
```

```
['simple', 'example', 'demonstrate', 'stop', 'word', 'removal', '.']
```

Regex

Regular expressions (regex) are a powerful tool in Python for pattern matching and text manipulation. The re module in Python provides support for regular expressions.

Writing regular expression (regex) patterns can seem complex at first, but understanding the basic components and common patterns can make it much easier. Here's a guide to help you write regex patterns effectively:

Basic Components of Regex

- Character Classes
  - [abc] # Matches 'a', 'b', or 'c'
  - [^abc] # Matches any character except 'a', 'b', or 'c'
  - [a-z] # Matches any lowercase letter
  - [A-Z] # Matches any uppercase letter
  - [0-9] # Matches any digit
  - \d # Matches any digit, equivalent to [0-9]
  - \D # Matches any non-digit
  - \w # Matches any alphanumeric character (word character)
  - \W # Matches any non-alphanumeric character
  - \s # Matches any whitespace character
  - \S # Matches any non-whitespace character
- Anchors
  - ^ # Matches the start of the string
  - $ # Matches the end of the string
  - \b # Matches a word boundary
  - \B # Matches a non-word boundary
- Quantifiers
  - * # Matches 0 or more occurrences
  - + # Matches 1 or more occurrences
  - ? # Matches 0 or 1 occurrence
  - {n} # Matches exactly n occurrences
  - {n,} # Matches n or more occurrences
  - {n,m} # Matches between n and m occurrences
- Grouping or Alteration

- (abc) # Groups 'abc' together
- (a|b) # Matches 'a' or 'b'

Examples

1. Email ID: [a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,}
2. Phone Number: \d{3}[-.]\d{3}[-.]\d{4}
3. URL: https?://(www.)?[a-zA-Z0-9._%+-]+.[a-zA-Z]{2,}
4. Date: \b\d{2}[-/]\d{2}[-/]\d{4}\b

The re.match() function checks if the beginning of a string matches the regex pattern.

In [ ]:
```
import re
pattern = r'hello'
text = 'hello world'

match = re.match(pattern, text)
if match:
    print('Match found:', match.group())
else:
    print('No match')
```

```
Match found: hello
```

The re.search() function searches the entire string for a match.

In [ ]:
```
pattern = r'world'
text = 'hello world'

search = re.search(pattern, text)
if search:
    print('Search found:', search.group())
else:
    print('No match')
```

```
Search found: world
```

The re.findall() function returns a list of all matches in the string.

In [ ]:
```
pattern = r'\d+'
text = 'There are 123 apples and 456 oranges.'

matches = re.findall(pattern, text)
print('All matches:', matches)
```

```
All matches: ['123', '456']
```

The re.split() function splits the string by occurrences of the pattern.

In [ ]:
```
pattern = r'\s+'
text = 'Split this text into words.'

splits = re.split(pattern, text)
print('Splits:', splits)
```

```
Splits: ['Split', 'this', 'text', 'into', 'words.']
```

The re.sub() function replaces occurrences of the pattern with a replacement string.

In [ ]:
```
pattern = r'apples'
text = 'I have apples and apples.'
replacement = 'oranges'

new_text = re.sub(pattern, replacement, text)
print('Replaced text:', new_text)
```

```
Replaced text: I have oranges and oranges.
```