

NLP Pipeline:

- We have seen some common NLP applications that we might encounter in everyday life. If we were asked to build such an application, think about how we would approach doing so at our organization. We would normally walk through the requirements and break the problem down into several sub-problems, then try to develop a step-by-step procedure to solve them.
- Since language processing is involved, we would also list all the forms of text processing needed at each step. This step-by-step processing of text is known as a pipeline. It is the series of steps involved in building any NLP model. These steps are common in most of the NLP project.
- Understanding some common procedures in any NLP pipeline will enable us to get started on any NLP problem encountered in the workplace. Laying out and developing a text-processing pipeline is seen as a starting point for any NLP application development process.
- Figure (1) shows the main components of a generic pipeline for modern-day, data-driven NLP system development. The key stages in the pipeline are as follows:
 1. Data acquisition
 2. Text cleaning
 3. Pre-processing
 4. Feature engineering
 5. Modeling
 6. Evaluation
 7. Deployment
 8. Monitoring and model updating

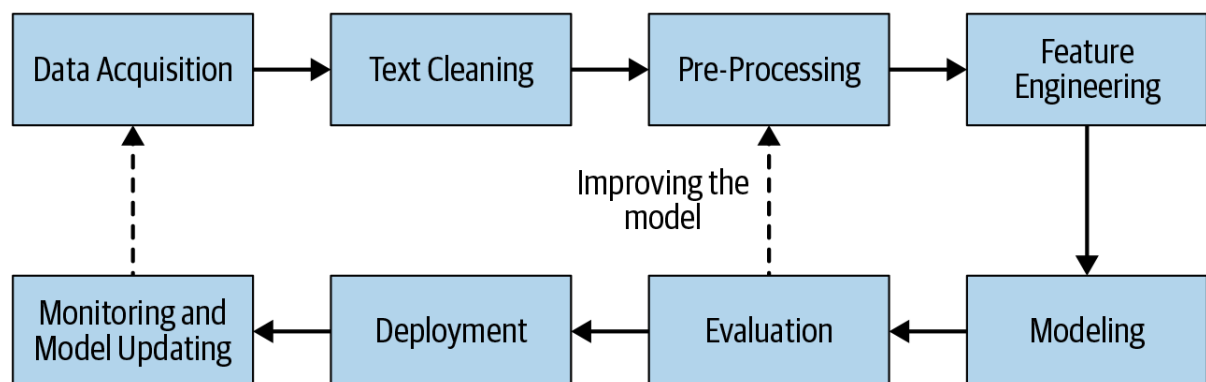


Figure 1: Generic NLP pipeline

- The first step in the process of developing any NLP system is to collect data relevant to the given task. The data we get is seldom clean, and this is where text cleaning comes into play.
- After cleaning, text data often has a lot of variations and needs to be converted into a canonical form. **This is done in the pre-processing step.** This is followed by feature engineering, where we carve out indicators that are most suitable for the task at hand.
- These indicators are converted into a format that is understandable by modeling algorithms. Then comes the modeling and evaluation phase, where we build one or more models and compare and contrast them using a relevant evaluation metric(s).
- Once the best model among the ones evaluated is chosen, we move toward deploying this model in production. Finally, we regularly monitor the performance of the model and, if need be, update it to keep up its performance.
- Note that, in the real world, the process may not always be linear as it's shown in the pipeline in Figure 1; it often involves going back and forth between individual steps (e.g., between feature extraction and modeling, modeling and evaluation, and so on).
- Also, there are loops in between, most commonly going from evaluation to pre-processing, feature engineering, modeling, and back to evaluation. There is also an overall loop that goes from monitoring to data acquisition, but this loop happens at the project level.
- Note that exact step-by-step procedures may depend on the specific task at hand. For example, a text-classification system may require a different feature extraction step compared to a text-summarization system.

Data Acquisition:

- Data is the heart of any ML/NLP system. Let's say we're asked to develop an NLP system **to identify whether an incoming customer query (for example, using a chat interface) is a sales inquiry or a customer care inquiry.** Depending on the type of query, it should be automatically routed to the right team. How can one go about building such a system? Well, the answer depends on the type and amount of data we have to work with.
- In an ideal setting, we'll have the required datasets with thousands—maybe even millions—of data points. In such cases, we don't have to worry about data acquisition. For example, in the scenario we just described, we have historic queries from previous years, which sales and support teams responded to.
- Further, the teams tagged these queries as sales, support, or other. So, not only do we have the data, but we also have the labels. However, in many AI projects, one is not so lucky. Let's look at what we can do in a less-than-ideal scenario.
- If we have little or no data, we can start by looking at patterns in the data that indicate if the incoming message is a sales or support query. We can then use regular expressions

and other heuristics to match these patterns to separate sales queries from support queries.

- We evaluate this solution by collecting a set of queries from both categories and calculating what percentage of the messages were correctly identified by our system. We would like to improve the system performance.
- Now we can start thinking about using NLP techniques. For this, we need labeled data, a collection of queries where each one is labeled with sales or support. How can we get such data?

1. Use a public dataset:

We could see if there are any public datasets available that we can leverage. Take a look at the compilation by **Nicolas Iderhoff** [<https://github.com/niderhoff/nlp-datasets>] or search Google's specialized search engine for datasets. If you find a suitable dataset that's similar to the task at hand, great! Build a model and evaluate. If not, then what?

2. Scrape data:

We could find a source of relevant data on the internet—for example, a consumer or discussion forum where people have posted queries (sales or support). Scrape the data from there and get it labeled by human annotators.

3. Product intervention:

In most industrial settings, AI models seldom exist by themselves. They're developed mostly to serve users via a feature or product. In all such cases, the AI team should work with the product team to collect more and richer data by developing better instrumentation in the product. **In the tech world, this is called product intervention.**

Product intervention is often the best way to collect data for building intelligent applications in industrial settings. Tech giants like Google, Facebook, Microsoft, Netflix, etc., have known this for a long time and have tried to collect as much data as possible from as many users as possible.

4. Data augmentation:

While instrumenting products is a great way to collect data, it takes time. Even if you instrument the product today, it can take anywhere between three to six months to collect a decent-sized, comprehensive dataset. So, can we do something in the meantime?

NLP has a bunch of techniques through which we can take a small dataset and use some tricks to create more data. These tricks are also called **data augmentation**, and they try to exploit language properties to create text that is syntactically similar to source text data. They may appear as hacks, but they work very well in practice.

Various techniques are:

Synonym replacement

Back translation

TF-IDF–based word replacement

Bi gram flipping

Replacing entities

5. Advanced techniques:

There are other advanced techniques and systems that can augment text data. Some of the notable ones are:

i) **Snorkel** [<https://www.snorkel.org/>] this is a system for building training data automatically, without manual labeling. Using Snorkel, a large training dataset can be “created”—without manual labeling—using heuristics and creating synthetic data by transforming existing data and creating new data samples.

ii) **Easy Data Augmentation (EDA) and NLPAug**

These two libraries are used to create synthetic samples for NLP. They provide implementation of various data augmentation techniques.

https://github.com/jasonwei20/eda_nlp

<https://arxiv.org/pdf/1901.11196.pdf>

<https://github.com/makcedward/nlpaug>

Text Extraction and Clean-up:

- Text extraction and clean up refers to the process of extracting raw text from the input data by removing all the other non-textual information, such as mark-up, metadata, etc., and converting the text to the required encoding format.
- Typically, this depends on the format of available data in the organization e.g., static data from PDF, HTML or text, some form of continuous data stream, etc.
- Text extraction is a standard data-wrangling step, and we don’t usually employ any NLP-specific techniques during this process.

1. HTML Parsing and Clean up:

- Suppose we’re working on a project where we’re building a forum search engine for programming questions. We’ve identified Stack Overflow as a source and decided to extract question and best-answer pairs from the website. How can we go through the text-extraction step in this case?
- If we observe the HTML markup of a typical Stack Overflow question page, we notice that questions and answers have special tags associated with them. We can utilize this information while extracting text from the HTML page.
- It’s more feasible to utilize existing **libraries** such as **Beautiful Soup** [<https://www.crummy.com/software/BeautifulSoup/>] and **Scrapy** [<https://scrapy.org/>], which provide a range of utilities to parse web pages.

- The following code snippet shows how to use BeautifulSoup to address the problem described here, extracting a question and its best-answer pair from a Stack Overflow web page:

```
from bs4 import BeautifulSoup
from urllib.request import urlopen
myurl = "https://stackoverflow.com/questions/415511/\
how-to-get-the-current-time-in-python"
html = urlopen(myurl).read()
soupified = BeautifulSoup(html, "html.parser")
question = soupified.find("div", {"class": "question"})
questiontext = question.find("div", {"class": "post-text"})
print("Question: \n", questiontext.get_text().strip())
answer = soupified.find("div", {"class": "answer"})
answertext = answer.find("div", {"class": "post-text"})
print("Best answer: \n", answertext.get_text().strip())
```

Here, we're relying on our knowledge of the structure of an HTML document to extract what we want from it. This code shows the output as follows:

Question:

What is the module/method used to get the current time?

Best answer:

Use:

```
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2009, 1, 6, 15, 8, 24, 78915)
```

```
>>> print(datetime.datetime.now())
2009-01-06 15:08:24.789150
```

And just the time:

```
>>> datetime.datetime.now().time()
datetime.time(15, 8, 24, 78915)
```

```
>>> print(datetime.datetime.now().time())
15:08:24.789150
```

See the documentation for more information.

To save typing, you can import the datetime object from the datetime module:

```
>>> from datetime import datetime
```

Then remove the leading datetime. from all of the above.

- In this example, we had a specific need: extracting a question and its answer. In some scenarios—for example, extracting postal addresses from web pages—we would get all the text (instead of only parts of it) from the web page first, before doing anything else.
- Typically, all HTML libraries have some function that can strip off all HTML tags and return only the content between the tags. But this often results in noisy output, and you may end up seeing a lot of JavaScript in the extracted content as well. In such cases, we should look to extract content between only those tags that typically contain text in web pages.

2. Unicode Normalization:

We may also encounter various **Unicode characters, including symbols, emojis, and other graphic characters**. A handful of Unicode characters are shown in Figure:

↑ U+2191	🙏 U+1F647	— U+2010	、 U+FF64	Θ U+0398	💚 U+1F49A	😊 U+263B	४ U+056E	ᳵ U+0C2C	ᳵ U+0CA0
Υ U+03B3	ᳵ U+12CE	💜 U+1F49C	μ U+03BC	🚀 U+1F680	🎵 U+266A	☺ U+FE36	· U+30FB	♡ U+10E6	” U+2036
☀ U+263C	। U+0964	○ U+26AC	ह U+0939	且 U+4E14	ब U+09F0	५ U+0F4F	% U+2030	テ U+30C7	↩ U+21A9
‘ U+2018	” U+2033	ℙ U+026A	ᳵ U+0DA2	😭 U+1F639	Δ U+0394	ù U+00F9	↩ U+27AB	ÿ U+0177	🙏 U+1F9D8

Figure 2: Unicode characters

- To parse such non-textual symbols and special characters, we use Unicode normalization. **This means that the text we see should be converted into some form of binary representation to store in a computer. This process is known as text encoding.**
- Ignoring encoding issues can result in processing errors further in the pipeline. There are several encoding schemes, and the default encoding can be different for different operating systems, especially when dealing with text in multiple languages, social media data, etc., we may have to convert between these encoding schemes during the text extraction process.

text = 'I love 🍕! Shall we book a 🚗 to gizza? '

```
Text = text.encode("utf-8")print(Text)
```

which outputs:

```
b' I love Pizza \xf0\x9f\x8d\x95! Shall we book a cab \xf0\x9f\x9a\x95 to get
pizza?'
```

This processed text is machine readable and can be used in downstream pipelines.

3. Spelling Correction

In the world of fast typing, incoming text data often has spelling errors. This can be prevalent in search engines, text-based chatbots deployed on mobile devices, social media, and many other sources. While we remove HTML tags and handle Unicode characters, this remains a unique problem that may hurt the linguistic understanding of the data, and shorthand text messages in social microblogs often hinder language processing and context understanding. Two such examples follow:

Shorthand typing: Hllo world! I am back!

Fat finger problem: I promise that I will not bresk the silence again!

While shorthand typing is prevalent in chat interfaces, fat-finger problems are common in search engines and are mostly unintentional. Despite our understanding of the problem, we don't have a robust method to fix this,

But we still can make good attempts to mitigate the issue. Microsoft released a REST API [<https://docs.microsoft.com/en-us/azure/cognitive-services/bing-spell-check/quickstarts/python>] that can be used in Python for potential spell checking:

Going beyond APIs, we can build our own spell checker using a huge dictionary of words from a specific language. A naive solution would be to look for all words that can be composed with minimal alteration (addition, deletion, substitution) to its constituent letters. For example, if "Hello" is a valid word that is already present in the dictionary, then the addition of "o" (minimal) to "Hllo" would make the correction.

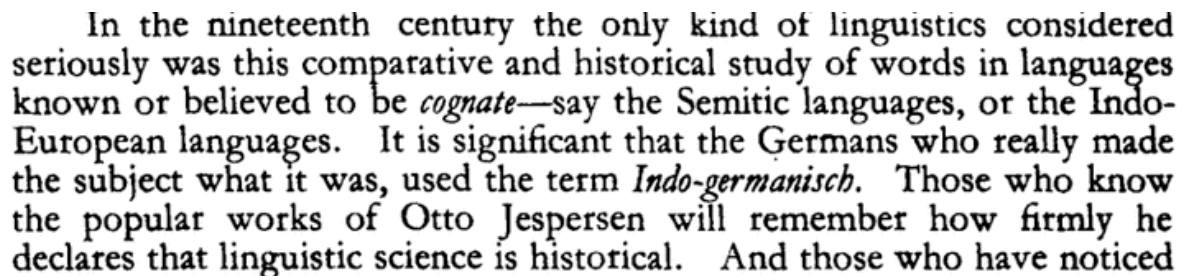
4. System-Specific Error Correction:

Consider another scenario where our dataset is in the form of a collection of PDF documents. The pipeline in this case starts with extraction of plain text from PDF documents. However, different PDF documents are encoded differently, and sometimes, we may not be able to extract the full text, or the structure of the text may get messed up.

If we need full text or our text has to be grammatical or in full sentences (e.g., when we want to extract relations between various people in the news based on newspaper text), this can impact our application. While there are several libraries, such as PyPDF [<https://github.com/mstamy2/PyPDF2>], PDFMiner [<https://github.com/pdfminer/pdfminer.six>],

etc., to extract text from PDF documents, they are far from perfect, and it's not uncommon to encounter PDF documents that can't be processed by such libraries.

- Another common source of textual data is **scanned documents**. Text extraction from scanned documents is typically done through optical character recognition (OCR), using libraries such as Tesseract [<https://github.com/tesseract-ocr/tesseract>] & [<https://pypi.org/project/pytesseract/>] Consider the example image shown in Figure below.



In the nineteenth century the only kind of linguistics considered seriously was this comparative and historical study of words in languages known or believed to be *cognate*—say the Semitic languages, or the Indo-European languages. It is significant that the Germans who really made the subject what it was, used the term *Indo-germanisch*. Those who know the popular works of Otto Jespersen will remember how firmly he declares that linguistic science is historical. And those who have noticed

Figure. An example of scanned text

The code snippet below shows how the Python library pytesseract can be used to extract text from this image:

```
from PIL import Image
from pytesseract import image_to_string
filename = "somefile.png"
text = image_to_string(Image.open(filename))
print(text)
```

This code will print the output as follows, where “\n” indicates a newline character:

```
`in the nineteenth century the only Kind of linguistics
considered\nseriously was this comparative and historical study
of words in languages\nknown or believed to Fe cognate—say the
Semitic languages, or the Indo-\nEuropean languages. It is
significant that the Germans who really made\nthe subject what
it was, used the term Indo-germanisch. Those who know\nthe
popular works of Otto Jespersen will remember how fitmly
he\ndeclares that linguistic science is historical. And those
who have noticed`
```

We notice that there are two errors in the output of the OCR system in this case. Depending on the quality of the original scan, OCR output can potentially have larger amounts of errors. How do we clean up this text before feeding it into the next stage of the pipeline?

One approach is to run the text through a spell checker such as **pyenchant** [<https://github.com/pyenchant/pyenchant>], which will identify misspellings and suggest some alternatives. More recent approaches use neural network architectures to train word/character-based language models, which are in turn used for correcting OCR text output based on the context [<https://github.com/KBNLresearch/ochre>].