**Preface**

Because of the nature of the project, we must work with a dataset that contains a large volume of text. The raw dataset we obtained suffers from the following issues:

* **Text granularity / structure:** In each row, the posts field actually concatenates 50 posts from a single user, separated by "|||".
* **Dirty characters and encoding:** The data contain URLs, HTML fragments, escape sequences, and leading single quotation marks.
* **Length and memory footprint:** Each row is about 7 k characters on average, with some rows exceeding 10 k characters.

These problems prevent the dataset from being used directly for analysis, so we must clean it first.

**Importing Modules and the Dataset**

1. **Import the required library**

import pandas as pd

This line imports the pandas library under the alias pd. Pandas is a powerful tool for data analysis and manipulation, especially well‑suited for reading and processing structured data such as CSV files.

1. **Import NLP‑related modules**

import re, string

from typing import List

import nltk

from nltk.tokenize import word\_tokenize

from nltk.corpus import stopwords, wordnet

from nltk import pos\_tag

from nltk.stem import WordNetLemmatizer

These statements import several NLP utilities:

* re, string: regular‑expression and character utilities for text processing.
* typing.List: type annotation for list data structures.
* The nltk suite: tokenisation (word\_tokenize), stop‑word handling (stopwords), part‑of‑speech tagging (pos\_tag), lemmatisation (WordNetLemmatizer), and more.

my\_nltk\_path = "Data"

nltk.data.path.append(my\_nltk\_path)

This sets the local NLTK resource path to the Data directory and appends it to the search path so that NLTK can load the required resources in the local environment.

import textstat

from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

* textstat: assesses text readability.
* SentimentIntensityAnalyzer: VADER’s sentiment‑analysis tool for assessing the affective polarity of text.

1. **Import modules for reading and writing external files**

import json

import pickle

import copy

* json: parsing and serialising JSON data.
* pickle: serialising and deserialising Python objects.
* copy: making shallow and deep copies of objects.

1. **Load the FastText language‑identification model**

import fasttext

lang\_model = fasttext.load\_model("lid.176.bin")

This loads Facebook’s FastText model lid.176.bin, which recognises 176 languages and returns language codes (e.g. "en", "zh", "es"). It is essential when pre‑processing multilingual data—especially when we need to standardise text to English via translation.

1. **Load a local large‑language model (LLM) and specify the model version**

import ollama

llm = "llama3.1:8b"

ollama provides an interface for running LLMs locally. Here we call the **LLaMA‑3** model with 8 billion parameters, a lightweight model used for on‑premise translation to improve privacy and efficiency when translating large corpora offline.

1. **Bring in the tqdm progress bar and enable Pandas integration**

from tqdm.auto import tqdm

tqdm.pandas()

TQDM is a popular Python progress‑bar library. tqdm.auto selects the most suitable display backend automatically (Jupyter‑friendly). Calling tqdm.pandas() enables progress bars for DataFrame.apply() and Series.apply() operations, giving real‑time feedback during long‑running tasks.

1. **Define an average‑value helper**

def ave(1):

return sum(1) / len(1)

A simple function ave is defined to calculate the average of a list of numeric values.

1. **Define the list of MBTI personality types**

MBTI\_types = [

"ISTJ", "ISFJ", "INFJ", "INTJ",

"ISTP", "ISFP", "INFP", "INTP",

"ESTP", "ESFP", "ENFP", "ENTP",

"ESTJ", "ESFJ", "ENFJ", "ENTJ"

]

A list named MBTI\_types is defined, containing all 16 Myers-Briggs personality types, to serve as the classification basis in subsequent text or label processing.

1. **Read the data and perform an initial split**

raw\_data = pd.read\_csv("Data\\twitter\_MBTI.csv", encoding="utf-8")

raw\_data.drop(columns="Unnamed: 0", inplace=True)

raw\_data.columns = ["posts", "type"]

for i in raw\_data.index:

temp = raw\_data.loc[i, "posts"]

temp = temp.split("|||")

raw\_data.loc[i, "posts"] = temp

First, the CSV file twitter\_MBTI.csv is loaded into a DataFrame. The automatically added index column ("Unnamed: 0") is removed, and the columns are renamed to standardise their labels. Each row’s posts field is then split on the delimiter "|||", converting the concatenated string into a list of separate posts for subsequent sentence‑level analysis.

1. **Load and merge the custom stop‑word list**

from custom\_stopwords import custom\_stopwords

stop\_words.update(custom\_stopwords)

custom\_stopwords is a task‑specific collection of terms—such as social‑media jargon ("lol", "omg", "http"), MBTI‑irrelevant proper nouns, or noise tokens—that should be excluded. Merging it into stop\_words ensures these words are filtered out during cleaning.

**Methods for Data Cleaning**

1. **Loading the contraction mapping**

with open(file="contractions.json", mode='r', encoding='utf-8') as f:

contractions\_map = json.load(f)

contractions.json maps common English contractions to their full forms (e.g. "don't" → "do not"), facilitating normalisation of user‑generated content.

1. **Constructor \_\_init\_\_**

def \_\_init\_\_(self, source=raw\_data):

self.data = source

Initialises the object with a DataFrame that contains a posts column in which each entry is a list of sentences.

1. **Remove user mentions and hashtags**

def remove\_mention\_and\_tag(self):

    def process\_removal(post):

        post\_without\_mention=[]

        for sentence in post:

            # Use re to scan and substitute

            post\_without\_mention.append(

                re.sub(

                    pattern=r'@\w+|#\w+',

                    repl=' ',

                    string=sentence

                )

            )

        return post\_without\_mention

    self.data["posts"]=self.data["posts"].apply(process\_removal)

This method defines an inner function process\_removal that scans each sentence in a post with the regular expression r'@\w+|#\w+' and replaces every mention or hashtag with a space, thereby stripping social‑media tags while preserving sentence order. The apply() call performs this operation on the entire posts column.

1. **Delete URL hyperlinks**

def remove\_url(self):

def process\_remove\_url(post):

post\_without\_url = []

for sentence in post:

post\_without\_url.append(

re.sub(

pattern=r'http\S+|www\S+|https\S+',

repl='',

string=sentence,

flags=re.MULTILINE

)

)

return post\_without\_url

self.data["posts"] = self.data["posts"].apply(process\_remove\_url)

The method removes any web‑page address (e.g. http://…, www.…), preventing links from contaminating downstream text analysis.

1. **Strip emoji characters**

def remove\_emoji(self):

def process\_remove\_emoji(post):

post\_without\_emoji=[]

for sentence in post:

# Use re to scan and substitute

emoji\_pattern=re.compile(

"["

"\U0001F600-\U0001F64F" # Emoticons

"\U0001F300-\U0001F5FF" # Miscellaneous Symbols and Pictographs

"\U0001F680-\U0001F6FF" # Transport and Map Symbols

"\U0001F1E0-\U0001F1FF" # Flags (iOS)

"\U00002702-\U000027B0" # Dingbats

"\U000024C2-\U0001F251" # Enclosed Characters, etc.

"\U0001f926-\U0001f937" # Supplemental Symbols and Pictographs

"\U00010000-\U0010ffff" # Broader range for some less common emojis

"]+", flags=re.UNICODE

)

The regular expression spans multiple Unicode ranges—covering most common and uncommon emojis such as facial expressions, gestures, vehicles, map icons, a wide variety of regional flags and pictographic symbols, and even supplementary characters beyond the Basic Multilingual Plane (BMP)—with flags=re.UNICODE enabled to ensure all Unicode characters are recognized.

post\_without\_emoji.append(

emoji\_pattern.sub(

repl=' ',

string=sentence

)

)

return post\_without\_emoji

self.data["posts"]=self.data["posts"].apply(process\_remove\_emoji)

For each sentence, the regex matches emojis and replaces every matched character with a single space (' '), preventing token concatenation noise while maintaining readability and analyzability; the process\_remove\_emoji function is then applied to the entire posts column via .apply(), bulk-removing emojis from every post and overwriting the original data column with the cleaned results.

1. **Expand English contractions**

**Static method text\_expand()**

@staticmethod

def text\_expand(original\_string, contraction\_mapping=contractions\_map):

Defines a static method text\_expand() that takes a single string (a sentence) as input and, by default, uses the pre‑loaded dictionary of contractions contractions\_map to convert contractions to their full forms.

contractions\_pattern = re.compile('({})'.format('|'.join(contraction\_mapping.keys())), flags=re.IGNORECASE | re.DOTALL)

This statement concatenates all the keys (i.e., every contraction) in the mapping dictionary into a regular‑expression pattern so that any occurrence of a contraction in the text can be captured. The flag re.IGNORECASE makes the match case‑insensitive, and re.DOTALL allows the pattern to span across line breaks.

def text\_mapping(text\_matched):

old\_text = text\_matched.group(0)

new\_text = contraction\_mapping.get(old\_text.lower())

if not new\_text:

new\_text = contraction\_mapping.get(old\_text)

if not new\_text:

return old\_text

return new\_text

The inner function text\_mapping() replaces each matched contraction with its full form. It first tries to match the lowercase version; if that fails, it tries the original case. If a replacement still cannot be found, it returns the original text unchanged, preventing information loss.

expanded\_string = contractions\_pattern.sub(repl=lambda m: text\_mapping(m), string=original\_string)

return expanded\_string

Using re.sub(), every match in the original string is replaced, and the expanded string is finally returned.

**Method expand\_contractions()**

def expand\_contractions(self):

def process\_expand\_contractions(original\_list):

for idx in range(len(original\_list)):

original\_list[idx] = Data\_to\_Clean.text\_expand(original\_list[idx])

return original\_list

self.data["posts"] = self.data["posts"].apply(lambda x: process\_expand\_contractions(x))

This method batch‑processes every posts entry in the DataFrame. Each posts entry is a list of strings, representing multiple sentences written by a user. The nested function process\_expand\_contractions() calls the static method text\_expand() on every sentence in the list, expanding all contractions to their full forms.

Finally, apply() is used to apply the processing to the entire column, achieving large‑scale contraction expansion. This greatly improves the consistency and semantic clarity of the data, which benefits subsequent tokenization, vectorization, and modeling stages.

1. **Convert to lower case**

def tolower(self):

def process\_tolower(post):

return [sentence.lower() for sentence in post]

self.data["posts"] = self.data["posts"].apply(process\_tolower)

Uniform lower‑casing prevents word‑frequency fragmentation due to case differences.

1. **Remove punctuation**

def remove\_punct(self):

def process\_remove\_punct(post):

post\_without\_punct = []

for sentence in post:

post\_without\_punct.append(

re.sub(pattern=r'[^a-zA-Z\s]', repl=' ', string=sentence)

)

return post\_without\_punct

self.data["posts"] = self.data["posts"].apply(process\_remove\_punct)

Non‑alphabetic characters are replaced by spaces, simplifying the text for language modelling.

1. **Delete blank strings**

def remove\_whitespace(self):

def process\_remove\_whitespace(post):

return [sentence for sentence in post if sentence.strip()]

self.data["posts"] = self.data["posts"].apply(process\_remove\_whitespace)

Sentences that are empty or contain only whitespace are discarded.

1. **Tokenization**

def totokens(self):

def process\_totokens(post):

post\_totokens = []

for sentence in post:

tokens = word\_tokenize(sentence)

post\_totokens.append(tokens)

return post\_totokens

self.data["posts"] = self.data["posts"].apply(process\_totokens)

word\_tokenize() splits each cleaned sentence into a list of tokens, providing the basic units for further NLP tasks.

1. **Remove stop‑words**

def remove\_stopwords(self):

def process\_remove\_stopwords(post):

stop\_words = set(stopwords.words("english"))

filtered\_post = []

for sentence in post:

filtered\_sentence = [word for word in sentence if word not in stop\_words]

filtered\_post.append(filtered\_sentence)

return filtered\_post

self.data["posts"] = self.data["posts"].apply(process\_remove\_stopwords)

Common high‑frequency, low‑information words are filtered out to emphasise lexical content relevant for modelling.

1. **Lemmatize**

def post\_lemmatize(self):

def process\_lemmatize(post):

def get\_wordnet\_postag(old\_postag):

if old\_postag.startswith('J'): return wordnet.ADJ

elif old\_postag.startswith('V'): return wordnet.VERB

elif old\_postag.startswith('N'): return wordnet.NOUN

elif old\_postag.startswith('R'): return wordnet.ADV

else: return wordnet.NOUN

Words are reduced to their dictionary forms based on POS tags, decreasing dimensionality and unifying inflected variants.

lemmatizer = WordNetLemmatizer()

lemmatized\_post = []

for tokens in post:

tagged\_tokens = pos\_tag(tokens)

lemmatized\_tokens = [lemmatizer.lemmatize(word, get\_wordnet\_postag(tag)) for word, tag in tagged\_tokens]

lemmatized\_post.append(lemmatized\_tokens)

return lemmatized\_post

self.data["posts"] = self.data["posts"].apply(process\_lemmatize)

The main function performs part-of-speech tagging on the tokenised text and, based on those tags, lemmatises each word (e.g., “running” → “run”), thereby unifying lexical representations and reducing dimensionality.

1. **Drop empty sentences**

def drop\_empty(self):

        def process\_drop(post):

            result=[sentence for sentence in post if sentence!=[]]

            return result

        self.data["posts"]=self.data["posts"].apply(process\_drop)

After cleaning, residual empty sentences are removed to maintain structural integrity.

**Methods for Data Analysis**

**1. Class Definition & Constructor**

class Data\_to\_Analyze(Data\_to\_Clean):

def \_\_init\_\_(self, type, source=raw\_data):

super().\_\_init\_\_(source)

self.data = self.data.loc[self.data["type"] == type].reset\_index(drop=True)

self.data\_to\_vec = None

self.basic\_identities = pd.Series({

"type": type,

# Number of sentences in a post

"sentence\_quantity": [],

"ave\_sentence\_quantity": None,

# Number of words in a post

"word\_count": [],

"ave\_word\_count": None,

# Ratio of upper‑case characters in a post

"upper\_ratio": [],

"ave\_upper\_ratio": None,

# Two readability indicators: Flesch Reading Ease and Gunning Fog Index

"reading\_ease": [],

"ave\_reading\_ease": None,

"GF\_index": [],

"ave\_GF\_index": None,

# Overall sentiment indicator (VADER)

"overall\_vader\_score": None

})

Defines a class Data\_to\_Analyze that inherits from Data\_to\_Clean.

* **Data subset** – The constructor calls super().\_\_init\_\_(source) to initialize the parent class, then filters the dataset so that only rows whose type column equals the target MBTI type remain (self.data).
* **Feature container** – basic\_identities is a pd.Series used to store a collection of text statistics: sentence count, word count, upper‑case ratio, readability metrics (Flesch Reading Ease and Gunning Fog Index) and an overall VADER sentiment score.

**2. Sentence Count**

def get\_sentence\_quantity(self):

for post in self.data["posts"].values:

self.basic\_identities["sentence\_quantity"].append(len(post))

self.basic\_identities["ave\_sentence\_quantity"] = ave(self.basic\_identities["sentence\_quantity"])

Iterates through every *post* (each post is already a list of sentences). The length of the list gives the number of sentences, which is appended to sentence\_quantity. Finally, the helper ave() computes their mean.

**3. Word Count**

def get\_word\_count(self):

for post in self.data["posts"].values:

total = 0

for sentence in post:

total += len(sentence.split(" "))

self.basic\_identities["word\_count"].append(total)

self.basic\_identities["ave\_word\_count"] = ave(self.basic\_identities["word\_count"])

For each post, it splits every sentence on whitespace to count words and sums them into total. The total per post is stored in word\_count, and the average is calculated afterwards.

**4. Upper‑Case Character Ratio**

def get\_upper\_ratio(self):

for post in self.data["posts"].values:

char\_count = 0

upper\_count = 0

for sentence in post:

for ch in sentence:

if ch.isalpha():

char\_count += 1

if ch.isupper():

upper\_count += 1

if char\_count:

self.basic\_identities["upper\_ratio"].append(upper\_count / char\_count)

self.basic\_identities["ave\_upper\_ratio"] = ave(self.basic\_identities["upper\_ratio"])

Traverses every character of every sentence, counting alphabetic characters (char\_count) and, among them, the upper‑case ones (upper\_count). The ratio per post is stored; the overall mean is then computed.

**5. Readability Metrics**

def get\_readability(self):

reading\_ease = []

GF\_idx = []

for post in self.data["posts"].values:

concatenated = post[0]

for idx in range(1, len(post)):

concatenated += post[idx]

reading\_ease.append(textstat.flesch\_reading\_ease(concatenated))

GF\_idx.append(textstat.gunning\_fog(concatenated))

self.basic\_identities["reading\_ease"] = reading\_ease

self.basic\_identities["ave\_reading\_ease"] = ave(reading\_ease)

self.basic\_identities["GF\_index"] = GF\_idx

self.basic\_identities["ave\_GF\_index"] = ave(GF\_idx)

Each post’s sentences are concatenated into a single string, after which textstat computes Flesch Reading Ease and Gunning Fog Index. Both per‑post values and their averages are stored.

**6. Helper: Concatenate a Full Post**

@staticmethod

def concatenate\_full\_post(post):

filtered = [s for s in post if not s.isspace()]

return "".join(filtered)

Removes purely blank sentences and joins the remainder—ready for downstream processing.

**7. VADER Sentiment Scoring**

analyzer = SentimentIntensityAnalyzer()

overall\_vader\_score = {'neg': 0.0, 'neu': 0.0, 'pos': 0.0, 'compound': 0.0}

* Instantiate NLTK’s SentimentIntensityAnalyzer to compute sentiment scores.
* Initialize overall\_vader\_score as an aggregate dictionary with the four sentiment components (neg, neu, pos, compound), which will be used to assess the text’s overall emotional state.

def addup\_score\_dict(new\_dict, base\_dict):

for key in base\_dict.keys():

base\_dict[key] += new\_dict[key]

This function adds the sentiment scores of a sentence (or post) to the target dictionary. It iterates over the keys and sums the corresponding values, allowing either local or global sentiment accumulation.

def ave\_score\_dict(base\_dict, n):

for key in base\_dict.keys():

base\_dict[key] /= n

This function divides each item in the cumulative score dictionary by the sample size n to obtain an average sentiment distribution. It is used for calculating the average score of a single post as well as the overall average for the dataset.

def process\_vader\_score(post):

post\_vader\_score = {'neg': 0.0, 'neu': 0.0, 'pos': 0.0, 'compound': 0.0}

for sentence in post:

addup\_score\_dict(analyzer.polarity\_scores(sentence), base\_dict=post\_vader\_score)

ave\_score\_dict(base\_dict=post\_vader\_score, n=len(post))

addup\_score\_dict(new\_dict=post\_vader\_score, base\_dict=overall\_vader\_score)

return post\_vader\_score

This nested function analyzes the sentiment of a single post (a list of sentences) through the following steps:

1. Initialize a score container for the post.
2. Traverse each sentence and obtain its sentiment scores with polarity\_scores().
3. Accumulate the sentence scores and divide by the number of sentences to compute the post’s average sentiment.
4. Add this post’s average score to the overall accumulator overall\_vader\_score.
5. Return the post’s sentiment score so it can be stored back in the dataset.

self.data["vader\_score"] = self.data["posts"].apply(process\_vader\_score)

This line applies process\_vader\_score to every post in self.data. The returned score dictionaries (neg, neu, pos, compound) are stored in a new column named vader\_score.

ave\_score\_dict(overall\_vader\_score, len(self.data["posts"]))

self.basic\_identities["overall\_vader\_score"] = overall\_vader\_score

After accumulating all post-level scores into overall\_vader\_score, divide by the total number of posts to obtain the average sentiment. Finally, record this overall score in the basic\_identities dictionary under the key "overall\_vader\_score" as the global emotional characteristic of the personality type.

**8. Automatically Detect Non‑English Sentences**

def locate\_str\_to\_translate(self):

result = []

for i in tqdm(self.data.index, desc="Locating strings of other languages..."):

for j in range(len(self.data.loc[i, "posts"])):

sentence = self.data.loc[i, "posts"][j]

* **Outer loop**: iterates over the entire DataFrame index, visiting each post sequentially.
* **Inner loop**: iterates through every sentence inside the current post (already stored as a list of sentences).
* tqdm provides a progress bar, allowing you to monitor the processing status when working with large‑scale data.

if len(sentence.split()) > 8:

Sets a lower bound of more than eight words for a sentence to be checked. Avoids language‑detection errors that commonly occur with very short sentences lacking context.

lang\_prediction = lang\_model.predict(

re.sub(r'\s+', ' ', sentence).strip(),

k=1)

* Cleans the sentence with a regular expression to collapse multiple spaces.
* Uses a fastText language‑identification model to predict the language; k=1 returns only the label with the highest probability.
* The output is a list of tuples in the form ('\_\_label\_\_xx', confidence).

if lang\_prediction[0][0] != "\_\_label\_\_en" and lang\_prediction[0][1] > 0.98:

The sentence is marked **“**needs translation**”** only if both conditions hold:

1. The predicted label is not English.
2. The model’s confidence is greater than 0.98.

result.append((i, j, re.sub(

pattern=r"\_\_\w+\_\_",

repl='',

string=lang\_prediction[0][0]

)))

Removes the "\_\_label\_\_" prefix, keeping only the ISO language code (e.g., "es", "de"). And appends a **triplet** (post\_index, sentence\_index, language\_code) to result.

self.locations = result

with open(f"Data/{self.basic\_identities["type"]}\_translate\_location.pkl", "wb") as f:

pickle.dump(result, f)

* Stores all non‑English sentence locations in the instance variable self.locations.
* Serializes the list to disk with pickle. The filename includes the MBTI type so that translation tasks can be handled in batches later.

**9. Translate with a Local LLM**

def translate\_str(self):

for coord in tqdm(self.locations, desc="Translating string into English..."):

sentence = self.data.loc[coord[0], "posts"][coord[1]]

self.data.loc[coord[0], "posts"][coord[1]] = ollama.generate(

model=llm,

prompt=(

f"Translate from {coord[2]} to English: \"{sentence}\"\n"

f"Output ONLY the translated text."

))['response']

Each non‑English sentence (identified in step 8) is translated in‑place by a locally hosted LLM (e.g. llama3.1:8b). The prompt specifies the source language and requests only the English rendition, preserving post order and structure.

**10. Filter Non‑English Sentences by Language ID**

def process\_drop(post, level=0.98):

filtered\_post = []

for sentence in post:

norm = re.sub(r"\s+", " ", sentence)

if len(norm.split()) < 6:

filtered\_post.append(norm)

continue

lang = lang\_model.predict(norm)

if lang[0][0] == '\_\_label\_\_en' and lang[0][1] > level:

filtered\_post.append(norm)

else:

# try lowercase version for robustness

lang = lang\_model.predict(norm.lower())

if lang[0][0] == '\_\_label\_\_en' and lang[0][1] > level:

filtered\_post.append(norm)

return filtered\_post

self.data["posts"] = self.data["posts"].apply(process\_drop)

Filters out non‑English sentences without translating them. Short sentences (< 6 words) are kept to avoid false negatives; longer sentences must pass an English‑language confidence threshold (level). A second check in lowercase mitigates errors caused by mixed casing or acronyms.

**Create a data‑processing method that encompasses the entire workflow**

**1. Initialize the Data Object & Remove URLs**

data = Data\_to\_Analyze(type=TYPE)

data.remove\_url()

data.remove\_mention\_and\_tag()

* Instantiate a Data\_to\_Analyze object to load the raw dataset for the specified personality type TYPE.
* Call remove\_url() and remove\_mention\_and\_tag() to strip out hyperlinks, user mentions (@username), and topic hashtags (#topic), eliminating social‑media noise.

**2. Extract Structured Text Features (Before Cleaning)**

data.get\_sentence\_quantity()

data.get\_word\_count()

data.get\_upper\_ratio()

data.get\_readability()

data.get\_vader\_score()

This step derives structural metrics from the **uncleaned** text:

* **get\_sentence\_quantity()** – counts the number of sentences in each post.
* **get\_word\_count()** – totals the words in each post.
* **get\_upper\_ratio()** – computes the ratio of uppercase letters to all alphabetic characters.
* **get\_readability()** – evaluates readability using two textstat metrics: *Flesch Reading Ease* and *Gunning Fog Index*.
* **get\_vader\_score()** – applies the VADER model for sentiment analysis, yielding composite sentiment scores.

*Important:* Run these functions before any cleaning steps such as contraction expansion or case conversion, since those operations alter the statistics.

**3. Continue Text‑Cleaning Tasks**

data.remove\_emoji()

data.remove\_whitespace()

data.drop\_non\_english(0.75)

data.expand\_contractions()

data.tolower()

data.remove\_punct()

data.remove\_whitespace()

data.totokens()

* **remove\_emoji()** – removes emoji and other non‑text graphical elements.
* **remove\_whitespace()** *(first pass)* – performs an initial whitespace cleanup.
* **drop\_non\_english(0.75)** – uses fastText to detect language and drops sentences that are not English when confidence is below *0.75*.
* **expand\_contractions()** – expands English contractions (e.g., "don’t" → "do not").
* **tolower()** – converts all text to lowercase, reducing word‑form variability.
* **remove\_punct()** – strips punctuation to facilitate vectorization or model processing.
* **remove\_whitespace()** *(second pass)* – clears residual extra spaces from earlier steps.
* **totokens()** – tokenizes the cleaned sentences into word lists (tokens) for downstream vectorization or language‑model tasks.

**4. Persist the Cleaned Data as a Binary File**

with open(f"Data\\cleaned\_data\\{TYPE}\_cleaned.pkl", "wb") as f:

pickle.dump(data, f)

Serializes the cleaned data object to disk in binary format with pickle, enabling rapid reuse for model training or further analysis.

**5. Run the Analysis in Batch for All MBTI Types**

for T in tqdm(MBTI\_types):

analyze\_data\_p1(T)

Iterates over MBTI\_types (assumed to include all 16 personality types) and executes analyze\_data\_p1() for each one, enabling efficient, consistent batch processing across the entire dataset.