1_individual_features

December 26, 2025

1 Individual features for AI detection

This notebook contains the code for developing the lightweight stylometric features for my baseline model and bias related features for my bias model.

With this code, each feature can be run on training data to detect if they seem worthwhile and how AI vs. human texts perfom with each feature. However, it requires a local copy of the PAN Task data from: https://zenodo.org/records/14962653

Of the features developed here, the following features were not moved into features.py for later use: - Baseline: TF-IDF - Bias: Emoji rate - Bias: Politeness - Bias: Political leaning - Bias: We vs. They

1.1 Baseline model

[1]:
```
# --- Set up environment ---
import pandas as pd
import spacy
import numpy as np

from collections import Counter
from collections import defaultdict
from pathlib import Path
from tqdm import tqdm
from sklearn.feature_extraction.text import TfidfVectorizer

tqdm.pandas()
```

[2]:
```
!python -m spacy download en_core_web_sm
nlp = spacy.load("en_core_web_sm")
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting en-core-web-sm==3.8.0
  Using cached https://github.com/explosion/spacy-
models/releases/download/en_core_web_sm-3.8.0/en_core_web_sm-3.8.0-py3-none-
any.whl (12.8 MB)
 Download and installation successful
You can now load the package via spacy.load('en_core_web_sm')
```

[ ]:
```
# Set paths
training_data = #Insert path to your training data here
```

[4]:
```
# Keep original and metadata
train_path = Path(training_data)
df_train = pd.read_json(train_path, lines=True)
df_train = df_train[["id", "text", "label", "genre", "model"]].copy()
df_train["label"] = df_train["label"].astype(int)
```

[5]:
```
# Apply spaCy to get all tokens, POS tags, etc.
df_train["spacy_doc"] = df_train["text"].progress_apply(nlp) # takes 40 to 90␣
 ↪min
```

```
100%|      | 23707/23707 [32:43<00:00, 12.08it/s]
```

[6]:
```
# --- Char-level TF-IDF vectorizer ---
char_vectorizer = TfidfVectorizer(
    analyzer='char',
    ngram_range=(3, 5),
    lowercase=True,
    max_features=50000)

# Apply to raw text
X_char = char_vectorizer.fit_transform(df_train["text"])

# Check output
print(f"Char TF-IDF shape: {X_char.shape}") # (num_texts, num_features)
print(char_vectorizer.get_feature_names_out()[:20]) # first 20 results (sanity␣
 ↪check)
```

```
Char TF-IDF shape: (23707, 50000)
[' "a' ' "a ' ' "ah' ' "ah,' ' "al' ' "an' ' "and' ' "b' ' "be' ' "bu'
 ' "but' ' "c' ' "ca' ' "co' ' "com' ' "d' ' "di' ' "do' ' "do ' ' "e']
```

[7]:
```
# --- Word-level TF-IDF vectorizer ---
# Remove numbers from text before vectorizing
df_train["text_no_numbers"] = df_train["text"].str.replace(r"\d+", "",␣
 ↪regex=True)

# Word-level TF-IDF vectorizer without numbers
word_vectorizer = TfidfVectorizer(
    analyzer='word',
    ngram_range=(1, 2),
    lowercase=True,
    token_pattern=r"(?u)\b\w+\b",
    max_features=50000
)
```

```
# Apply to raw text without numbers
X_word = word_vectorizer.fit_transform(df_train["text_no_numbers"])

# Check output
print(f"Word TF-IDF shape: {X_word.shape}") # (num_texts, num_features)
print(word_vectorizer.get_feature_names_out()[:20]) # first 20 results (sanity␣
 ↪check)
```

```
Word TF-IDF shape: (23707, 50000)
['a' 'a a' 'a baby' 'a bachelor' 'a back' 'a backdrop' 'a bad' 'a balance'
 'a balanced' 'a ball' 'a balm' 'a band' 'a bank' 'a bargain' 'a barrier'
 'a basket' 'a bastion' 'a battle' 'a battleground' 'a beacon']
```

[ ]:
```
# --- Sentence and word length ---
def extract_length_features(doc):
    # Sentences (filtered for valid tokens)
    sent_lengths = [
        len([t for t in sent if not t.is_punct])
        for sent in doc.sents
        if len(sent) > 0
    ]

    # Words (exclude punctuation)
    words = [t for t in doc if not t.is_punct]
    word_lengths = [len(t.text) for t in words]

    # Compute stats
    return pd.Series({
        "mean_sent_len": np.nanmean(sent_lengths) if sent_lengths else 0,
        "std_sent_len":  np.nanstd(sent_lengths) if len(sent_lengths) > 1 else␣
 ↪0,
        "mean_word_len": np.nanmean(word_lengths) if word_lengths else 0,
        "std_word_len":  np.nanstd(word_lengths) if len(word_lengths) > 1 else␣
 ↪0,
    })

# Apply to spaCy text
length_features = df_train["spacy_doc"].progress_apply(extract_length_features)

# Combine features with labels
length_features_labeled = length_features.copy()
length_features_labeled["label"] = df_train["label"] # 0=Human, 1=AI

# Look at class-wise means
print(length_features_labeled.groupby("label").mean())
print(length_features_labeled.groupby("label").std())
```

```
100%|      | 23707/23707 [00:34<00:00, 694.22it/s]
```

```
       mean_sent_len  std_sent_len  mean_word_len  std_word_len
label
0          23.492785     14.494070       4.326734      2.316890
1          23.612674      8.521213       4.998295      2.716548
       mean_sent_len  std_sent_len  mean_word_len  std_word_len
label
0           7.974506      6.180024       0.337722      0.249901
1           7.900751      6.590442       0.654281      0.570322
```

[ ]:
```python
# --- Stop words ---
def extract_stopword_rate(doc):
    tokens = [t for t in doc if not t.is_punct]
    num_tokens = len(tokens)
    num_stopwords = sum(t.is_stop for t in tokens)

    stopword_ratio = num_stopwords / num_tokens if num_tokens > 0 else 0
    return pd.Series({
        "stopword_ratio": stopword_ratio
    })

# Apply to spaCy text
stopword_features = df_train["spacy_doc"].progress_apply(extract_stopword_rate)

# Combine features with labels
stopword_features_labeled = stopword_features.copy()
stopword_features_labeled["label"] = df_train["label"] # 0=Human, 1=AI

# Look at class-wise means
print(stopword_features_labeled.groupby("label").mean())
print(stopword_features_labeled.groupby("label").std())
```

```
100%|      | 23707/23707 [00:09<00:00, 2594.98it/s]
       stopword_ratio
label
0            0.559110
1            0.467336
       stopword_ratio
label
0            0.053475
1            0.059773
```

[ ]:
```python
# --- Punctuation ---
# Define punctuation marks
punct_marks = [",", ".", ";", ":", "?", "!", "'", '"', "(", ")", "-", "…"]
```

```
def extract_punct_rates(text):
    total_chars = len(text)
    counts = {f"punct_{p}": text.count(p) / total_chars * 1000 if total_chars >␣
 ↪0 else 0
              for p in punct_marks}
    return pd.Series(counts)

# Apply to raw text
punct_features = df_train["text"].progress_apply(extract_punct_rates)

# Combine features with labels
punct_features_labeled = punct_features.copy()
punct_features_labeled["label"] = df_train["label"] # 0=Human, 1=AI

# Look at class-wise means
print(punct_features_labeled.groupby("label").mean())
print(punct_features_labeled.groupby("label").std())
```

```
100%|      | 23707/23707 [00:04<00:00, 5414.65it/s]
         punct_,   punct_.   punct_;   punct_:   punct_?   punct_!   punct_'  \
label
0      13.406649  8.800357  1.192362  0.255844  0.788814  0.673093  2.168841
1      12.626805  7.935055  0.132706  0.127867  0.360970  0.141684  2.033503

         punct_"   punct_(   punct_)   punct_-   punct_…
label
0      3.173604  0.144225  0.144691  2.926624  0.009305
1      2.155658  0.097837  0.097856  0.960964  0.033617
         punct_,   punct_.   punct_;   punct_:   punct_?   punct_!   punct_'  \
label
0      3.757094  3.114569  1.100631  0.423927  0.948124  0.961134  3.611919
1      4.225618  2.766029  0.294887  0.346481  0.673576  0.463970  2.313164

         punct_"   punct_(   punct_)   punct_-   punct_…
label
0      5.021078  0.387232  0.388396  2.756644  0.080972
1      3.117307  0.328473  0.327992  1.063832  0.214208
```

[ ]:
```
# --- Casing ratios ---
def extract_char_class_ratios(text):
    total_chars = len(text)
    total_tokens = len(text.split())

    # Avoid division by zero
```

```
    if total_chars == 0:
        return pd.Series({k: 0 for k in ["upper_ratio", "title_ratio",␣
↪"digit_ratio", "space_ratio"]})

    upper_ratio = sum(c.isupper() for c in text) / total_chars
    digit_ratio = sum(c.isdigit() for c in text) / total_chars
    space_ratio = sum(c.isspace() for c in text) / total_chars

    # Titlecase = first letter uppercase, rest lowercase
    title_ratio = sum(w.istitle() for w in text.split()) / total_tokens if␣
↪total_tokens > 0 else 0

    return pd.Series({
        "upper_ratio": upper_ratio,
        "title_ratio": title_ratio,
        "digit_ratio": digit_ratio,
        "space_ratio": space_ratio
    })

# Apply to raw text
casing_features = df_train["text"].progress_apply(extract_char_class_ratios)

# Combine features with labels
casing_features_labeled = casing_features.copy()
casing_features_labeled["label"] = df_train["label"] # 0=Human, 1=AI

# Look at class-wise means
print(casing_features_labeled.groupby("label").mean())
print(casing_features_labeled.groupby("label").std())
```

```
100%|      | 23707/23707 [00:20<00:00, 1148.21it/s]
       upper_ratio  title_ratio  digit_ratio  space_ratio
label
0         0.021202     0.103604     0.001172     0.179693
1         0.018649     0.098261     0.001543     0.161593
       upper_ratio  title_ratio  digit_ratio  space_ratio
label
0         0.009640     0.037458     0.003460     0.009878
1         0.009516     0.043080     0.003507     0.013396
```

[ ]:
```
# --- Type-Token ratio ---
def extract_ttr(doc):
    tokens = [t.text.lower() for t in doc if t.is_alpha]
    num_tokens = len(tokens)
    num_types = len(set(tokens))
```

```
    ttr = num_types / num_tokens if num_tokens > 0 else 0
    return pd.Series({"ttr": ttr})

# Apply to spaCy text
ttr_feature = df_train["spacy_doc"].progress_apply(extract_ttr)

# Combine features with labels
ttr_features_labeled = ttr_feature.copy()
ttr_features_labeled["label"] = df_train["label"] # 0=Human, 1=AI

# Look at class-wise means
print(ttr_features_labeled.groupby("label").mean())
print(ttr_features_labeled.groupby("label").std())
```

```
100%|      | 23707/23707 [00:14<00:00, 1623.63it/s]
            ttr
label
0      0.473008
1      0.526680
            ttr
label
0      0.047294
1      0.065601
```

[ ]:
```
# --- UPOS distribution --
# Fixed list of tags to extract
upos_tags = ['NOUN', 'VERB', 'ADJ', 'ADV', 'PRON', 'ADP', 'DET', 'CCONJ',␣
 ↪'SCONJ', 'NUM', 'AUX', 'INTJ', 'PART', 'PROPN']

def extract_upos_freq(doc):
    tags = [t.pos_ for t in doc if not t.is_punct and not t.is_space]
    total = len(tags)

    tag_counts = Counter(tags)
    freqs = {
        f"upos_{tag}": tag_counts.get(tag, 0) / total if total > 0 else 0
        for tag in upos_tags
    }
    return pd.Series(freqs)

# Apply to spaCy text
upos_features = df_train["spacy_doc"].progress_apply(extract_upos_freq)

# Combine features with labels
```

```
upos_features_labeled = upos_features.copy()
upos_features_labeled["label"] = df_train["label"] # 0=Human, 1=AI

# Look at class-wise means
print(upos_features_labeled.groupby("label").mean())
print(upos_features_labeled.groupby("label").std())
```

```
100%|      | 23707/23707 [00:11<00:00, 2093.04it/s]
       upos_NOUN  upos_VERB  upos_ADJ  upos_ADV  upos_PRON  upos_ADP  \
label
0       0.181809   0.128340  0.072362  0.053546   0.121642  0.115584
1       0.242442   0.128684  0.087093  0.038426   0.082271  0.125813

       upos_DET  upos_CCONJ  upos_SCONJ  upos_NUM  upos_AUX  upos_INTJ  \
label
0      0.097427    0.040256    0.025845  0.009726  0.065546   0.002768
1      0.111099    0.032557    0.019728  0.005655  0.037501   0.000715

       upos_PART  upos_PROPN
label
0       0.029937    0.054544
1       0.028116    0.059417
       upos_NOUN  upos_VERB  upos_ADJ  upos_ADV  upos_PRON  upos_ADP  \
label
0       0.035080   0.019883  0.018824  0.014838   0.041499  0.019767
1       0.036582   0.018593  0.027313  0.015071   0.042357  0.019193

       upos_DET  upos_CCONJ  upos_SCONJ  upos_NUM  upos_AUX  upos_INTJ  \
label
0      0.021138    0.011393    0.008735  0.009170  0.017575   0.003732
1      0.024107    0.011087    0.008388  0.008431  0.016646   0.001662

       upos_PART  upos_PROPN
label
0       0.009967    0.036257
1       0.011188    0.045347
```

[ ]:
```
# --- Max 5-gram repetition rate ---
def compute_5gram_repetition(doc):
    # Use alpha-only tokens (optional: can include all if you prefer)
    tokens = [t.text.lower() for t in doc if not t.is_space]
    n = 5
    total_tokens = len(tokens)

    if total_tokens < n:
```

```
        return pd.Series({"rep_5gram_ratio": 0.0})

    # Count all 5-grams
    counts = defaultdict(int)
    for i in range(total_tokens - n + 1):
        fivegram = tuple(tokens[i:i+n])
        counts[fivegram] += 1

    # Find all repeated 5-grams
    repeated_spans = set()
    for i in range(total_tokens - n + 1):
        fivegram = tuple(tokens[i:i+n])
        if counts[fivegram] > 1:
            repeated_spans.update(range(i, i+n))

    # Compute repetition ratio
    rep_ratio = len(repeated_spans) / total_tokens if total_tokens > 0 else 0
    return pd.Series({"rep_5gram_ratio": rep_ratio})

# Apply to spaCy text
rep_features = df_train["spacy_doc"].progress_apply(compute_5gram_repetition)
df_train["token_count"] = df_train["spacy_doc"].apply(lambda doc: len([t for t␣
 ↪in doc if not t.is_space]))

# Combine features with labels
rep_features_labeled = rep_features.copy()
rep_features_labeled["label"] = df_train["label"] # 0=Human, 1=AI

# Look at class-wise means
print(rep_features_labeled.groupby("label").mean())
print(rep_features_labeled.groupby("label").std())
```

```
100%|      | 23707/23707 [00:26<00:00, 901.18it/s]
```

```
       rep_5gram_ratio
label
0             0.022223
1             0.032773
       rep_5gram_ratio
label
0             0.038228
1             0.068343
```

[ ]:
```
# --- Self-similarity ---
def compute_self_similarity(doc, k=200):
    # Use clean lowercase tokens
    tokens = [t.text.lower() for t in doc if t.is_alpha]
    total = len(tokens)
```

```
    half = total // 2
    first_half = tokens[:half]
    second_half = tokens[half:]

    # Count top-k unigrams in each half
    first_top = set([w for w, _ in Counter(first_half).most_common(k)])
    second_top = set([w for w, _ in Counter(second_half).most_common(k)])

    # Compute Jaccard similarity
    intersection = first_top & second_top
    union = first_top | second_top

    sim = len(intersection) / len(union) if union else 0.0
    return pd.Series({"self_sim_jaccard": sim})

# Apply to spaCy docs
selfsim_features = df_train["spacy_doc"].progress_apply(compute_self_similarity)

# Combine features with labels
selfsim_features_labeled = selfsim_features.copy()
selfsim_features_labeled["label"] = df_train["label"] # 0=Human, 1=AI

# Look at class-wise means
print(selfsim_features_labeled.groupby("label").mean())
print(selfsim_features_labeled.groupby("label").std())
```

```
100%|      | 23707/23707 [00:17<00:00, 1339.47it/s]
       self_sim_jaccard
label
0              0.180080
1              0.167926
       self_sim_jaccard
label
0              0.038875
1              0.047826
```

1.2 Bias model

[ ]:
```
# --- Set up environment ---
import nltk
import pandas as pd
import numpy as np
import re

from nltk.sentiment import SentimentIntensityAnalyzer
```

```
from nltk.tokenize import sent_tokenize
from pathlib import Path
from torch import argmax
from torch.nn.functional import softmax
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from tqdm import tqdm

tqdm.pandas()

nltk.download("vader_lexicon")
nltk.download("punkt")
nltk.download('punkt_tab')
```

[17]:
```
!python -m spacy download en_core_web_sm
nlp = spacy.load("en_core_web_sm")
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting en-core-web-sm==3.8.0
  Downloading https://github.com/explosion/spacy-
models/releases/download/en_core_web_sm-3.8.0/en_core_web_sm-3.8.0-py3-none-
any.whl (12.8 MB)
     ---------------------------------------- 0.0/12.8 MB ? eta -:--:--
     - -------------------------------------- 0.5/12.8 MB 21.8 MB/s eta 0:00:01
     ------------------------------ ------- 10.5/12.8 MB 68.1 MB/s eta 0:00:01
     ---------------------------------------- 12.8/12.8 MB 30.8 MB/s  0:00:00
  Download and installation successful
You can now load the package via spacy.load('en_core_web_sm')
```

[ ]:
```
# Set paths
training_data = #Insert path to your training data here
```

[20]:
```
# Keep original and metadata
train_path = Path(training_data)
df_bias = pd.read_json(train_path, lines=True)
df_bias = df_bias[["id", "text", "label", "genre", "model"]].copy()
df_bias["label"] = df_bias["label"].astype(int)
```

[ ]:
```
# --- Sentiment analysis ---
vader = SentimentIntensityAnalyzer()
def compute_sentiment_features(text):
    sentences = sent_tokenize(text)
    if not sentences:
        return pd.Series({"sentiment_mean": 0.0, "sentiment_var": 0.0})

    scores = [vader.polarity_scores(s)["compound"] for s in sentences]
    return pd.Series({
        "sentiment_mean": np.mean(scores),
        "sentiment_var": np.var(scores)
```

```
    })

# Apply to dataframe
sentiment_features = df_bias["text"].progress_apply(compute_sentiment_features)

# Add results into df
df_bias["sentiment_mean"] = sentiment_features["sentiment_mean"]
df_bias["sentiment_var"] = sentiment_features["sentiment_var"]

# Check output
df_bias.groupby("label")[["sentiment_mean", "sentiment_var"]].agg(["mean",␣
 ↪"var"]) # 0=Human, 1=AI
```

```
100%|      | 23707/23707 [02:48<00:00, 140.60it/s]
```

[ ]:

	sentiment_mean		sentiment_var	
	mean	var	mean	var
label				
0	0.089463	0.022290	0.165208	0.003891
1	0.126481	0.048832	0.167544	0.004575

[ ]:
```
# --- Emoji rate ---
emojis = {" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", ":)", ":-)", ":(", ":
 ↪-("}

def compute_emoji_rate(text):
    total_chars = len(text)
    if total_chars == 0:
        return 0.0
    emoji_count = sum(text.count(e) for e in emojis)
    return emoji_count / total_chars

# Apply to dataframe
emoji_rate = df_bias["emoji_rate"] = df_bias["text"].
 ↪progress_apply(compute_emoji_rate)

# Check output
df_bias.groupby("label")["emoji_rate"].agg(["mean", "var"]) # 0=Human, 1=AI
```

```
  0%|          | 0/23707 [00:00<?, ?it/s]
100%|      | 23707/23707 [00:00<00:00, 84958.39it/s]
```

[ ]:

	mean	var
label		
0	2.492696e-08	5.654936e-12
1	3.231006e-08	1.524779e-11

[ ]:
```
# --- Profanity and insults ---
# Load profanity / insult lexicon (one word per line)
path = #Insert path to toxic-words-dictionary.txt here (can be downloaded from␣
 ↪github repository)

def load_profanity_lexicon(path):
    with open(path, "r", encoding="cp1252") as f:
        return {line.strip().lower() for line in f if line.strip()}

profanity_lexicon = load_profanity_lexicon(path)

def compute_profanity_rate(text):
    tokens = re.findall(r"\b\w+\b", text.lower())
    if not tokens:
        return 0.0
    count = sum(token in profanity_lexicon for token in tokens)
    return count / len(tokens)

# Apply to dataframe
profanity_rate = df_bias["profanity_rate"] = df_bias["text"].
 ↪progress_apply(compute_profanity_rate)

# Check output
stats_by_label = df_bias.groupby("label")["profanity_rate"].describe()
stats_by_label = stats_by_label[["count", "mean", "std", "max"]]
print(stats_by_label) # 0=Human, 1=AI
```

```
  0%|          | 0/23707 [00:00<?, ?it/s]
100%|      | 23707/23707 [00:07<00:00, 3178.49it/s]
         count      mean       std       max
label
0       9101.0  0.004868  0.004586  0.068000
1      14606.0  0.003404  0.005084  0.091716
```

[ ]:
```
# --- Politeness and impoliteness (few results found in PAN Task data) --
# Define lexicons
politeness_lexicon = {
    "polite_markers": [
        # Gratitude / appreciation
        "please", "thank", "thanks", "thank you", "appreciate", "grateful",␣
 ↪"many thanks",
        "much obliged", "i am grateful", "i appreciate", "that would be great",
        "thankfully", "cheers",

        # Apologies / softeners
```

```
        "sorry", "excuse me", "pardon", "forgive me", "i apologize", "my␣
↪apologies",
        "hope you don't mind", "if you don't mind", "if you please", "beg your␣
↪pardon",

        # Indirect / hedging requests
        "if you could", "would you mind", "may i", "could you", "would you",␣
↪"might you",
        "it would be great if", "i was wondering if", "i would appreciate it␣
↪if",
        "do you think you could", "perhaps", "possibly", "hopefully", "maybe",␣
↪"might be",
        "i think", "i suppose", "it seems", "i believe", "in my opinion", "from␣
↪my point of view",

        # Positive politeness / friendliness
        "nice", "kind", "wonderful", "lovely", "dear", "friendly", "helpful",
        "great job", "well done", "good work", "amazing", "fantastic",␣
↪"brilliant",
        "pleased", "glad", "delighted", "it's a pleasure", "so kind of you",
        "how are you", "hope you're well", "thank you so much"
    ],

    "impolite_markers": [
        # Direct insults / name-calling
        "stupid", "idiot", "moron", "dumb", "fool", "loser", "jerk", "bastard",␣
↪"trash",
        "worthless", "lazy", "ignorant", "pathetic", "useless", "annoying",␣
↪"nonsense",

        # Aggressive imperatives / dismissals
        "shut up", "shut it", "shut your mouth", "get lost", "go away", "leave␣
↪me alone",
        "mind your own business", "don't bother", "don't you dare", "stop␣
↪talking",
        "listen", "you must", "you have to", "you should", "you better", "you␣
↪need to",

        # Sarcastic / condescending tones
        "obviously", "clearly", "of course you would", "as if", "whatever",␣
↪"yeah right",
        "sure thing", "you think so", "what a joke", "get real",␣
↪"unbelievable", "ridiculous",

        # Hostility / emotional aggression
        "hate", "disgusting", "gross", "awful", "terrible", "horrible", "nasty",
```

```
        "idiotic", "moronic", "pathetic", "shame on you", "who cares", "no one␣
 ↪cares",
        "screw you", "go to hell"
    ]
}

def compute_politeness_from_lexicon(text, lexicon):
    text_lower = text.lower()
    tokens = re.findall(r"\b\w+\b", text_lower)

    if not tokens:
        return pd.Series({
            "polite_rate": 0.0,
            "impolite_rate": 0.0,
            "politeness_score": 0.0
        })

    polite_words = {w for w in lexicon["polite_markers"] if " " not in w}
    polite_phrases = [w for w in lexicon["polite_markers"] if " " in w]

    impolite_words = {w for w in lexicon["impolite_markers"] if " " not in w}
    impolite_phrases = [w for w in lexicon["impolite_markers"] if " " in w]

    # single-word matches
    polite_rate = sum(t in polite_words for t in tokens) / len(tokens)
    impolite_rate = sum(t in impolite_words for t in tokens) / len(tokens)

    # phrase matches
    polite_rate += sum(p in text_lower for p in polite_phrases) / len(tokens)
    impolite_rate += sum(p in text_lower for p in impolite_phrases) /␣
 ↪len(tokens)

    return pd.Series({
        "polite_rate": polite_rate,
        "impolite_rate": impolite_rate,
        "politeness_score": polite_rate - impolite_rate
    })

# Apply to dataframe
politeness_features = df_bias["text"].progress_apply(
    compute_politeness_from_lexicon, lexicon=politeness_lexicon
)

# Add into dataframe
df_bias["polite_rate"] = politeness_features["polite_rate"]
df_bias["impolite_rate"] = politeness_features["impolite_rate"]
df_bias["politeness_score"] = politeness_features["politeness_score"]
```

```
# Check output
stats_by_label_politeness = df_bias.groupby("label")["politeness_score"].
 ↪describe()
stats_by_label_politeness = stats_by_label_politeness[["count", "mean", "std",␣
 ↪"max"]]

print(stats_by_label_politeness) # 0=Human, 1=AI
```

```
100%|      | 23707/23707 [00:18<00:00, 1284.83it/s]
```

label	count	mean	std	max
0	9101.0	0.002168	0.003547	0.023288
1	14606.0	0.001213	0.002822	0.029762

[ ]:
```
# --- Political leaning ---
# Load model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("launch/POLITICS")
model = AutoModelForSequenceClassification.from_pretrained("matous-volf/
 ↪political-leaning-politics")
model.eval()

# Prediction helper function
def classify_political_leaning(text):
    try:
        tokens = tokenizer(text[:512], return_tensors="pt")  # limit to first␣
 ↪512 tokens for speed
        output = model(**tokens)
        logits = output.logits
        leaning_idx = argmax(logits, dim=1).item()
        probs = softmax(logits, dim=1)
        score = probs[0, leaning_idx].item()

        label_map = {0: "left", 1: "center", 2: "right"}
        leaning_label = label_map.get(leaning_idx, "unknown")

        return pd.Series({
            "stance_pred": leaning_label,
            "stance_score": score,
            "prob_left": probs[0, 0].item(),
            "prob_center": probs[0, 1].item(),
            "prob_right": probs[0, 2].item()
        })

    except Exception:
```

```
        return pd.Series({
            "stance_pred": "error",
            "stance_score": 0.0,
            "prob_left": 0.0,
            "prob_center": 0.0,
            "prob_right": 0.0
        })

# Apply to dataset
stance_features = df_bias["text"].progress_apply(classify_political_leaning)
df_bias["stance_pred"] = stance_features["stance_pred"]
df_bias["stance_score"] = stance_features["stance_score"]
df_bias["prob_left"] = stance_features["prob_left"]
df_bias["prob_center"] = stance_features["prob_center"]
df_bias["prob_right"] = stance_features["prob_right"]

# Check output of political stance leanings
print("\nAverage stance probabilities by label:")
print(df_bias.groupby("label")[["prob_left", "prob_center", "prob_right"]].
 ↪mean()) # 0=Human, 1=AI
```

```
 27%|          | 6334/23707 [14:18<43:29,  6.66it/s]  Token indices sequence
length is longer than the specified maximum sequence length for this model (726
> 512). Running this sequence through the model will result in indexing errors
100%|      | 23707/23707 [47:50<00:00,  8.26it/s]


Average stance probabilities by label:
       prob_left  prob_center  prob_right
label
0       0.612678     0.258223    0.129099
1       0.648738     0.204894    0.146231
```

[ ]:
```
# --- Identity terms ---
# Define lexicons
identity_lexicons = {
    "gender": {
        "he", "him", "his", "she", "her", "hers", "woman", "man",
        "male", "female", "boy", "girl", "mother", "father",
        "sister", "brother"
    },

    "race_ethnicity": {
        "white", "black", "asian", "latino", "hispanic",
        "african", "european", "indian", "arab"
    },
```

```
    "religion": {
        "christian", "muslim", "jewish", "hindu", "buddhist",
        "islam", "christianity", "judaism"
    },

    "nationality": {
        "american", "british", "german", "french", "spanish",
        "russian", "chinese", "japanese", "italian", "canadian"
    },

    "orientation": {
        "gay", "lesbian", "bisexual", "transgender", "queer", "lgbt"
    },

    "disability": {
        "blind", "deaf", "autistic", "disabled", "mental",
        "handicapped", "wheelchair"
    },

    "age": {
        "child", "kid", "youth", "teen", "adult", "elderly",
        "senior", "old", "young"
    }
}

# Compute rates
def compute_identity_term_rates(text, lexicons=identity_lexicons):
    tokens = re.findall(r"\b\w+\b", text.lower())
    total = len(tokens)

    if total == 0:
        return pd.Series({f"{k}_identity_rate": 0.0 for k in lexicons})

    features = {}
    for category, words in lexicons.items():
        count = sum(t in words for t in tokens)
        features[f"{category}_identity_rate"] = count / total

    return pd.Series(features)

# Apply to dataframe
identity = df_bias["text"].progress_apply(compute_identity_term_rates)

for col in identity.columns:
    df_bias[col] = identity[col]
```

```python
# Check output
print("\nIdentity Term Frequency Comparison:\n")

for col in identity.columns:
    print(f"=== {col} ===")
    print(df_bias.groupby("label")[col].describe(), "\n") # 0=Human, 1=AI
```

```
100%|      | 23707/23707 [00:17<00:00, 1332.44it/s]


Identity Term Frequency Comparison:

=== gender_identity_rate ===
         count      mean       std  min       25%       50%       75%  \
label
0       9101.0  0.046038  0.027518  0.0  0.025316  0.045817  0.064759
1      14606.0  0.032907  0.029982  0.0  0.004425  0.027192  0.054130

            max
label
0      0.153734
1      0.156383

=== race_ethnicity_identity_rate ===
         count      mean       std  min  25%  50%       75%       max
label
0       9101.0  0.000970  0.002300  0.0  0.0  0.0  0.001418  0.047809
1      14606.0  0.000637  0.002567  0.0  0.0  0.0  0.000000  0.061617

=== religion_identity_rate ===
         count      mean       std  min  25%  50%  75%       max
label
0       9101.0  0.000123  0.000918  0.0  0.0  0.0  0.0  0.031936
1      14606.0  0.000186  0.001573  0.0  0.0  0.0  0.0  0.041543

=== nationality_identity_rate ===
         count      mean       std  min  25%  50%  75%      max
label
0       9101.0  0.000528  0.001618  0.0  0.0  0.0  0.0  0.03858
1      14606.0  0.000597  0.002091  0.0  0.0  0.0  0.0  0.04000

=== orientation_identity_rate ===
         count      mean       std  min  25%  50%  75%       max
label
0       9101.0  0.000108  0.000836  0.0  0.0  0.0  0.0  0.047486
1      14606.0  0.000100  0.001995  0.0  0.0  0.0  0.0  0.085799

=== disability_identity_rate ===
```

```
         count      mean       std  min  25%  50%  75%       max
label
0       9101.0  0.000133  0.000747  0.0  0.0  0.0  0.0  0.029210
1      14606.0  0.000179  0.001212  0.0  0.0  0.0  0.0  0.037249

=== age_identity_rate ===
         count      mean       std  min  25%       50%       75%       max
label
0       9101.0  0.003165  0.003475  0.0  0.0  0.002304  0.004573  0.028065
1      14606.0  0.001499  0.002664  0.0  0.0  0.000000  0.002188  0.042904
```

[ ]:
```python
# --- Hedging and modality ---
# Define lexicons
hedge_lexicon = {
    "modals": [
        "may", "might", "could", "should", "would",
        "can", "shall", "ought",
        "possibly", "conceivably"
    ],

    "adverbs": [
        "perhaps", "probably", "possibly", "apparently", "likely",
        "evidently", "presumably", "seemingly", "maybe",
        "hopefully", "arguably", "theoretically",
        "hypothetically", "roughly", "approximately",
        "almost", "virtually", "essentially", "generally",
        "typically", "usually", "frequently", "often",
        "mostly", "largely", "somewhat", "partially"
    ],

    "verbs": [
        "seem", "appear", "suggest", "imply", "assume", "believe",
        "think", "guess", "suspect", "estimate", "suppose",
        "consider", "imagine", "indicate", "hint", "propose",
        "speculate", "predict", "infer", "presume",
    ],

    "phrases": [
        "it seems", "it appears", "it is possible", "it is likely",
        "there is a chance", "it could be", "it might be",
        "to some extent", "in a way", "in some cases",
        "in certain respects", "it may be that",
        "one could argue", "one might think",
        "it could appear", "it could suggest"
    ]
```

```
}

# Compute hedging frequency
def compute_hedge_rate(text, lexicon):
    text_lower = text.lower()
    tokens = re.findall(r"\b\w+\b", text_lower)
    total_tokens = len(tokens)
    if total_tokens == 0:
        return 0.0

    # Single-word matches
    single_words = set(lexicon["modals"] + lexicon["adverbs"] +␣
 ↪lexicon["verbs"])
    count_single = sum(t in single_words for t in tokens)

    # Multi-word phrase matches
    count_phrases = sum(p in text_lower for p in lexicon["phrases"])

    return (count_single + count_phrases) / total_tokens

# Apply to dataframe
hedge_rate = df_bias["hedge_rate"] = df_bias["text"].progress_apply(lambda x:␣
 ↪compute_hedge_rate(x, hedge_lexicon))

# Check output
stats_by_label = df_bias.groupby("label")["hedge_rate"].describe()
print("\nHedging / Modality Rate Comparison:")
print(stats_by_label) # 0=Human, 1=AI
```

```
100%|      | 23707/23707 [00:08<00:00, 2763.92it/s]


Hedging / Modality Rate Comparison:
         count      mean       std  min       25%       50%       75%  \
label
0       9101.0  0.014677  0.007716  0.0  0.009160  0.013636  0.019074
1      14606.0  0.009545  0.006938  0.0  0.004444  0.008299  0.013201

            max
label
0      0.068807
1      0.063325
```

[ ]:
```
# --- Categorical statements ---
# Define lexicons
categorical_lexicon = [
```

```
    # Universal quantifiers
    "always", "never", "ever", "all", "none",
    "every", "everyone", "everything", "everybody",
    "nobody", "nothing", "noone", "no-one",

    # Absolutes / extremes
    "completely", "entirely", "absolutely", "totally",
    "utterly", "purely", "perfectly", "fully",
    "definitely", "certainly", "inevitably", "unquestionably",
    "undeniably", "unconditionally",

    # Extremeness / maximality
    "forever", "eternal", "eternally", "infinite", "infinitely",
    "permanent", "permanently", "final", "finally",
    "ultimate", "ultimately", "guaranteed", "guarantee",

    # Strong modal certainty
    "must", "cannot", "can't", "will", "won't",
    "always", "never",

    # Strong evaluative absolutes
    "best", "worst", "only", "everyone", "nobody",
    "universal", "universally", "all-time"
]

# Compute rate of categorical statements
def compute_categorical_rate(text, lexicon=categorical_lexicon):
    text_lower = text.lower()
    tokens = re.findall(r"\b\w+\b", text_lower)
    total_tokens = len(tokens)
    if total_tokens == 0:
        return 0.0

    return sum(t in lexicon for t in tokens) / total_tokens

# Apply to dataframe
categorical_rate = df_bias["categorical_rate"] = df_bias["text"].
 ↪progress_apply(compute_categorical_rate)

# Check output
stats_by_label = df_bias.groupby("label")["categorical_rate"].describe()
print("\nCategorical words Rate Comparison:")
print(stats_by_label) # 0=Human, 1=AI
```

```
100%|      | 23707/23707 [00:17<00:00, 1355.07it/s]
```

```
Categorical words Rate Comparison:
```

```
          count      mean       std  min       25%       50%       75%  \
label
0        9101.0  0.014699  0.007259  0.0  0.009472  0.014006  0.019118
1       14606.0  0.008067  0.005673  0.0  0.004000  0.007174  0.011050

             max
label
0       0.059084
1       0.049505
```

[ ]:
```python
# --- Assertiveness ---
# Define lexicons
assertive_lexicon = [
    # Strong factual/claim verbs
    "prove", "proves", "proved",
    "demonstrate", "demonstrates", "demonstrated",
    "show", "shows", "shown",
    "confirm", "confirms", "confirmed",
    "establish", "establishes", "established",
    "verify", "verifies", "verified",

    # Strong certainty adverbs
    "clearly", "certainly", "definitely", "undeniably",
    "obviously", "evidently", "unquestionably",
    "undoubtedly", "incontrovertibly", "irrefutably",
    "surely", "decisively",

    # Strong necessity / obligation signals
    "must", "cannot", "can't", "will", "won't",
    "have to", "has to", "need to",

    # Strong evaluative certainty
    "without a doubt", "beyond doubt", "beyond question",
    "it is clear", "it is certain",
    "everyone knows", "it is obvious",

    # Rhetorical emphasis
    "in fact", "the truth is", "the reality is"
]

# Compute rate of assertive words
def compute_assertive_rate(text, lexicon=assertive_lexicon):
    text_lower = text.lower()
    tokens = re.findall(r"\b\w+\b", text_lower)
    total_tokens = len(tokens)
    if total_tokens == 0:
```

```
        return 0.0

    return sum(t in lexicon for t in tokens) / total_tokens

# Apply to dataframe
assertive_rate = df_bias["assertive_rate"]   = df_bias["text"].
 ↪progress_apply(compute_assertive_rate)

# Check output
stats_by_label = df_bias.groupby("label")["assertive_rate"].describe()
print("\nAssertive words Rate Comparison:")
print(stats_by_label) # 0=Human, 1=AI
```

```
100%|      | 23707/23707 [00:16<00:00, 1475.17it/s]


Assertive words Rate Comparison:
         count      mean       std  min       25%       50%       75%  \
label
0       9101.0  0.004606  0.004173  0.0  0.001504  0.003683  0.006319
1      14606.0  0.002974  0.003760  0.0  0.000000  0.001939  0.004298

            max
label
0      0.038405
1      0.041958
```

[ ]:
```
# --- Subjectivity ---
# Define lexicons
subjective_lexicon = [
    # subjective adjectives
    "good", "bad", "terrible", "wonderful", "awful", "amazing",
    "horrible", "excellent", "disgusting", "fantastic", "stupid",
    "brilliant", "ridiculous", "shocking", "tragic", "beautiful",
    "ugly", "unfair", "unjust", "biased", "corrupt",

    # subjective verbs (opinions, evaluations)
    "believe", "think", "feel", "guess", "suspect", "argue",
    "claim", "insist", "support", "oppose", "criticize",
    "praise", "endorse", "blame", "doubt", "prefer", "hate",
    "love", "enjoy", "fear", "worry"
]

# Compute rate of subjective words
def compute_subjectivity_score(text, lexicon=subjective_lexicon):
    text_lower = text.lower()
```

```
    tokens = re.findall(r"\b\w+\b", text_lower)
    total = len(tokens)
    if total == 0:
        return 0.0
    return sum(t in lexicon for t in tokens) / total

# Apply to dataframe
df_bias["subjectivity_score"] = df_bias["text"].
 ↪progress_apply(compute_subjectivity_score)

# Check output
stats_by_label = df_bias.groupby("label")["subjectivity_score"].describe()
print("\nSubjective words Rate Comparison:")
print(stats_by_label) # 0=Human, 1=AI
```

```
100%|      | 23707/23707 [00:14<00:00, 1599.12it/s]


Subjective words Rate Comparison:
         count      mean       std  min       25%       50%       75%  \
label
0       9101.0  0.005252  0.004020  0.0  0.002418  0.004478  0.007386
1      14606.0  0.003241  0.003598  0.0  0.000000  0.002342  0.004745

            max
label
0      0.033141
1      0.037528
```

[ ]:
```
# --- Positivity and negativity --
# Load lexicons

def load_lexicon(path):
    """Loads a lexicon file (one word per line, allows comment lines)."""
    with open(path, "r") as f:
        return {line.strip().lower() for line in f
                if line.strip() and not line.startswith(";")}

positive_lexicon = load_lexicon(#Insert path to positive-words-dictionary.txt␣
 ↪here (can be downloaded from github repository))
negative_lexicon = load_lexicon(#Insert path to negative-words-dictionary.txt␣
 ↪here (can be downloaded from github repository))

# Compute rates
def compute_emotional_tone_from_lexicons(text, pos_lex, neg_lex):
    tokens = re.findall(r"\b\w+\b", text.lower())
```

```
    if not tokens:
        return pd.Series({"pos_rate": 0.0, "neg_rate": 0.0, "polarity_score": 0.
 ↪0})
    pos_rate = sum(t in pos_lex for t in tokens) / len(tokens)
    neg_rate = sum(t in neg_lex for t in tokens) / len(tokens)
    return pd.Series({
        "pos_rate": pos_rate,
        "neg_rate": neg_rate,
        "polarity_score": pos_rate - neg_rate
    })

# Apply to dataframe
emotional_tone = df_bias["text"].
 ↪progress_apply(compute_emotional_tone_from_lexicons,␣
 ↪pos_lex=positive_lexicon, neg_lex=negative_lexicon)

# Add into df
df_bias["pos_rate"] = emotional_tone["pos_rate"]
df_bias["neg_rate"] = emotional_tone["neg_rate"]
df_bias["polarity_score"] = emotional_tone["polarity_score"]

# Check output
stats_by_label_emotional_tone = df_bias.groupby("label")[["pos_rate",␣
 ↪"neg_rate", "polarity_score"]].describe()

# Flatten hierarchical columns
stats_by_label_emotional_tone.columns = ['_'.join(col).strip() for col in␣
 ↪stats_by_label_emotional_tone.columns.values]
print(stats_by_label_emotional_tone) # 0=Human, 1=AI
```

```
100%|      | 23707/23707 [00:13<00:00, 1732.76it/s]
       pos_rate_count  pos_rate_mean  pos_rate_std  pos_rate_min  \
label
0              9101.0       0.033994      0.012791           0.0
1             14606.0       0.043895      0.018656           0.0

       pos_rate_25%  pos_rate_50%  pos_rate_75%  pos_rate_max  neg_rate_count  \
label
0          0.024969      0.033105      0.041729      0.106814          9101.0
1          0.030588      0.042105      0.055307      0.144000         14606.0

       neg_rate_mean  …  neg_rate_75%  neg_rate_max  polarity_score_count  \
label                 …
0           0.030889  …      0.038690      0.099190                9101.0
1           0.041190  …      0.053977      0.161392               14606.0

       polarity_score_mean  polarity_score_std  polarity_score_min  \
```

```
label
0                 0.003105            0.019209           -0.083700
1                 0.002705            0.032198           -0.136076

       polarity_score_25%  polarity_score_50%  polarity_score_75%  \
label
0               -0.008982            0.004045            0.015588
1               -0.019063            0.002398            0.023644

       polarity_score_max
label
0                0.086556
1                0.144000

[2 rows x 24 columns]
```

[ ]:
```python
# --- We vs. They (dropped because unstable) ---
# Define lexicons
we_pronouns = {"we", "us", "our", "ours", "ourselves"}
they_pronouns = {"they", "them", "their", "theirs", "themselves"}

# Compute rate of pronouns
def compute_we_they_features(text):
    text_lower = text.lower()
    tokens = re.findall(r"\b\w+\b", text_lower)
    total = len(tokens)
    if total == 0:
        return pd.Series({
            "we_pronoun_rate": 0.0,
            "they_pronoun_rate": 0.0,
            "we_they_ratio": 0.0
        })

    we_count = sum(t in we_pronouns for t in tokens)
    they_count = sum(t in they_pronouns for t in tokens)

    we_rate = we_count / total
    they_rate = they_count / total

    # avoid division by zero
    if they_count == 0:
        ratio = float(we_count > 0)  # 1 if we>0, else 0
    else:
        ratio = we_count / they_count

    return pd.Series({
```

```
        "we_pronoun_rate": we_rate,
        "they_pronoun_rate": they_rate,
        "we_they_ratio": ratio
    })

# Apply to dataframe
we_they = df_bias["text"].progress_apply(compute_we_they_features)
df_bias["we_pronoun_rate"] = we_they["we_pronoun_rate"]
df_bias["they_pronoun_rate"] = we_they["they_pronoun_rate"]
df_bias["we_they_ratio"] = we_they["we_they_ratio"]

# Check output
print("\nWe/They Feature Comparison:\n")

for col in ["we_pronoun_rate", "they_pronoun_rate", "we_they_ratio"]:
    print(f"=== {col} ===")
    print(df_bias.groupby("label")[col].describe(), "\n") # 0=Human, 1=AI
```

```
100%|      | 23707/23707 [00:12<00:00, 1892.05it/s]


We/They Feature Comparison:

=== we_pronoun_rate ===
         count      mean       std  min  25%       50%       75%       max
label
0       9101.0  0.004938  0.007160  0.0  0.0  0.002725  0.006466  0.113456
1      14606.0  0.005450  0.009087  0.0  0.0  0.001560  0.007159  0.080675

=== they_pronoun_rate ===
         count      mean       std  min       25%       50%       75%  \
label
0       9101.0  0.008490  0.007725  0.0  0.003040  0.006460  0.011561
1      14606.0  0.008355  0.008644  0.0  0.002568  0.005985  0.011282

            max
label
0      0.109677
1      0.091190

=== we_they_ratio ===
         count      mean       std  min  25%       50%  75%   max
label
0       9101.0  0.998576  2.128660  0.0  0.0  0.333333  1.0  48.0
1      14606.0  1.040896  2.447203  0.0  0.0  0.166667  1.0  37.0
```