DE-AI CIPHER

DECODING THE LANGUAGE OF MACHINES

By: Hardik Sharma

Introduction

1. The rise of AI language models has blurred the lines between machine-generated and human-generated text.

2. This project aims to explore the possibility of separating AI-generated text from human-written text by analyzing a comprehensive set of various linguistic and statistical metrics.

3. By examining features such as perplexity, stylometric patterns, syntactic structures, semantic coherence, and others, the project will evaluate which metrics are most effective in distinguishing between the two types of text.

4. The ultimate goal is to deepen our understanding of the characteristics that differentiate AI-generated content from human writing and assess the feasibility of accurately separating them.

Motivation

The ability to differentiate between AI-generated and human-written text is increasingly important for several reasons:

1. In an era where information is abundant, verifying the authenticity of content is crucial for maintaining trust in digital communications.
2. AI-generated texts can be used to spread fake news or propaganda. Identifying such content helps mitigate the impact of disinformation.
3. As AI tools become accessible, there is a risk of misuse in academic settings. Detecting AI-generated submissions is essential to uphold academic standards.
4. Understanding how AI-generated text differs from human writing contributes to responsible AI development and deployment.

Dataset

DAIGT | Catch The AI (Link): This data consists of different LLMs , such as: Mistral-7B(v1&v2) , Llama 70b , Falcon180b ,GPT(3.5 & 4), Claude.

Training Records: 25969 ,Validation Records: 2730 and Testing Records: 2730

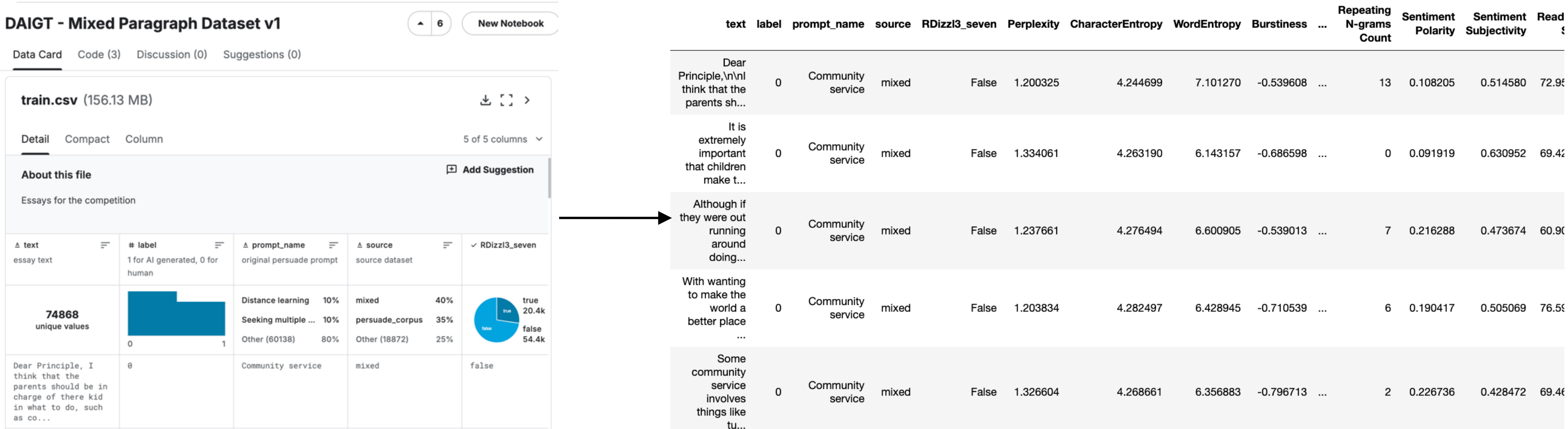
DAIGT - Mixed Paragraph Dataset v1 (Link):

All Records: 74868 unique records

LLM - Detect AI Generated Text Dataset (Link): The dataset comprises of a mixture of 28,000 student-written essays and essays generated by a variety of LLMs.

All Records: 27340 unique records

Methodology



text	label	prompt_name	source	RDizzl3_seven	Perplexity	CharacterEntropy	WordEntropy	Burstiness	...	Repeating N-grams Count	Sentiment Polarity	Sentiment Subjectivity	Read
Dear Principle,\n\nI think that the parents sh...	0	Community service	mixed	False	1.200325	4.244699	7.101270	-0.539608	...	13	0.108205	0.514580	72.9
It is extremely important that children make t...	0	Community service	mixed	False	1.334061	4.263190	6.143157	-0.686598	...	0	0.091919	0.630952	69.4
Although if they were out running around doing...	0	Community service	mixed	False	1.237661	4.276494	6.600905	-0.539013	...	7	0.216288	0.473674	60.9
With wanting to make the world a better place ...	0	Community service	mixed	False	1.203834	4.282497	6.428945	-0.710539	...	6	0.190417	0.505069	76.5
Some community service involves things like tu...	0	Community service	mixed	False	1.326604	4.268661	6.356883	-0.796713	...	2	0.226736	0.428472	69.4

1. Take raw text data

2. Calculate features/LLM Metric Score for each text record

3. Preprocess the resultant data
- Remove any missing value
- Normalize all the features

4. Divide the data into Training, Validation and Testing

5. Apply various machine learning algorithms on the validation dataset and store the one with best performance. Apply that model on the test data.

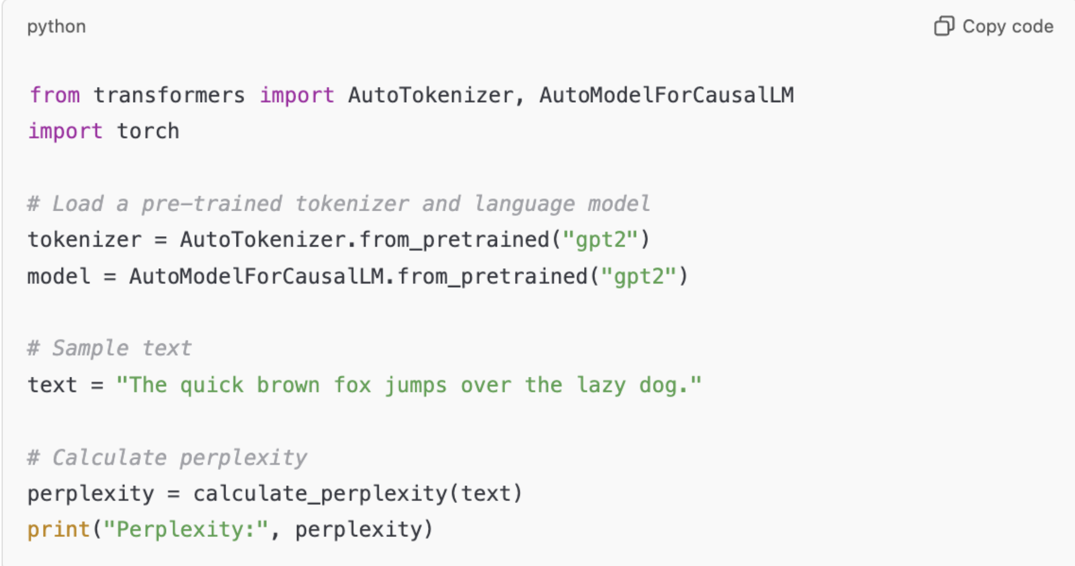
LLM Metrics : Perplexity (1/17)

In this function we calculate the **perplexity** of a given text using a pre-trained language model. (in my case, I used bert-base-uncased). Perplexity is a measure of how well a language model predicts the text. Lower perplexity indicates the text is more predictable or aligned with the language model's training, while higher perplexity suggests the text is harder to predict.

Explanation:

1. **Tokenization**: The text is tokenized into numerical inputs using a pre-trained tokenizer.
2. **Compute loss:** The model calculates the loss based on how well it predicts the given text.
3. **Perplexity calculation:** Perplexity is calculated as the exponential of the loss: e^{loss}

Example:



Output:

LLM Metrics : Entropy (2/17)

In this function, we calculate the **word-wise entropy** and **character-wise entropy** of a text. Each entropy metric measures the randomness or diversity of word/character usage by evaluating the probability distribution of words/characters in the text. Higher entropy indicates greater variation in word/character choice, while lower entropy suggests repetitive or predictable language.

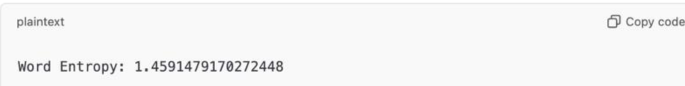
Key Points:

1. **Word-wise entropy** captures the variation in word choice (e.g., high entropy in diverse vocabulary).
2. **Character-wise entropy** captures the diversity in character usage (e.g., high entropy in text with varied characters).

Example:

```python
from collections import Counter
import math

# Sample text
text = "hello world hello world hello universe"

# Calculate word entropy
word_entropy = calculate_word_entropy(text)
print("Word Entropy:", word_entropy)
```

Output:



```plaintext
Word Entropy: 1.4591479170272448
```

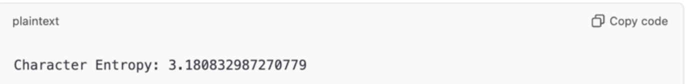
Explanation:

1. **Word probabilities:**
   - Words: `["hello", "world", "universe"]`
   - Frequencies: `{"hello": 3, "world": 2, "universe": 1}`
   - Probabilities: $p_{\text{hello}} = 3/6$, $p_{\text{world}} = 2/6$, $p_{\text{universe}} = 1/6$
2. **Entropy formula:**

Example:

```python
# Sample text
text = "hello world"

# Calculate character entropy
char_entropy = calculate_char_entropy(text)
print("Character Entropy:", char_entropy)
```

Output:



```plaintext
Character Entropy: 3.180832987270779
```

Explanation:

1. **Character probabilities:**
   - Characters: `["h", "e", "l", "o", " ", "w", "r", "d"]`
   - Frequencies: `{"h": 1, "e": 1, "l": 3, "o": 2, " ": 1, "w": 1, "r": 1, "d": 1}`
   - Probabilities: E.g., $p_l = 3/11$
2. **Entropy formula:**

$$H = -\sum p_i \cdot \log_2(p_i) \approx 3.18$$

LLM Metrics : Burstiness (3/17)

In this function we calculate the **burstiness** of words in a text. Burstiness measures the unevenness or irregularity in the occurrence of words across the text. If a word appears in quick bursts (closely clustered positions) rather than evenly spread, it has higher burstiness. The metric is calculated based on the intervals between occurrences of the same word.

Key Points:

1. **Low Burstiness:** Words are evenly spaced, resulting in a score that is close to 0.
2. **High Burstiness:** Words cluster together, creating irregular intervals and a higher burstiness score.

Example:

```python
from collections import defaultdict
import numpy as np

# Sample text
text = "dog cat dog cat dog dog cat cat cat"

# Calculate burstiness
burstiness_score = calculate_burstiness(text)
print("Burstiness Score:", burstiness_score)
```

Output:

```plaintext
Burstiness Score: 0.08333333333333331
```

Explanation:

1. **Track word positions:**
   - Words: `["dog", "cat", "dog", "cat", "dog", "dog", "cat", "cat", "cat"]`
   - Positions: `{"dog": [0, 2, 4, 5], "cat": [1, 3, 6, 7, 8]}`
2. **Calculate intervals for each word:**
   - For "dog": Intervals = `[2, 2, 1]`
   - For "cat": Intervals = `[2, 3, 1, 1]`
3. **Compute burstiness:**
   - For "dog":
     - Mean interval: 1.67
     - Standard deviation: 0.47
     - Burstiness = $\frac{\text{std}-\text{mean}}{\text{std}+\text{mean}}$
   - Similar for "cat."
4. **Average burstiness:**
   - Final score: 0.0833

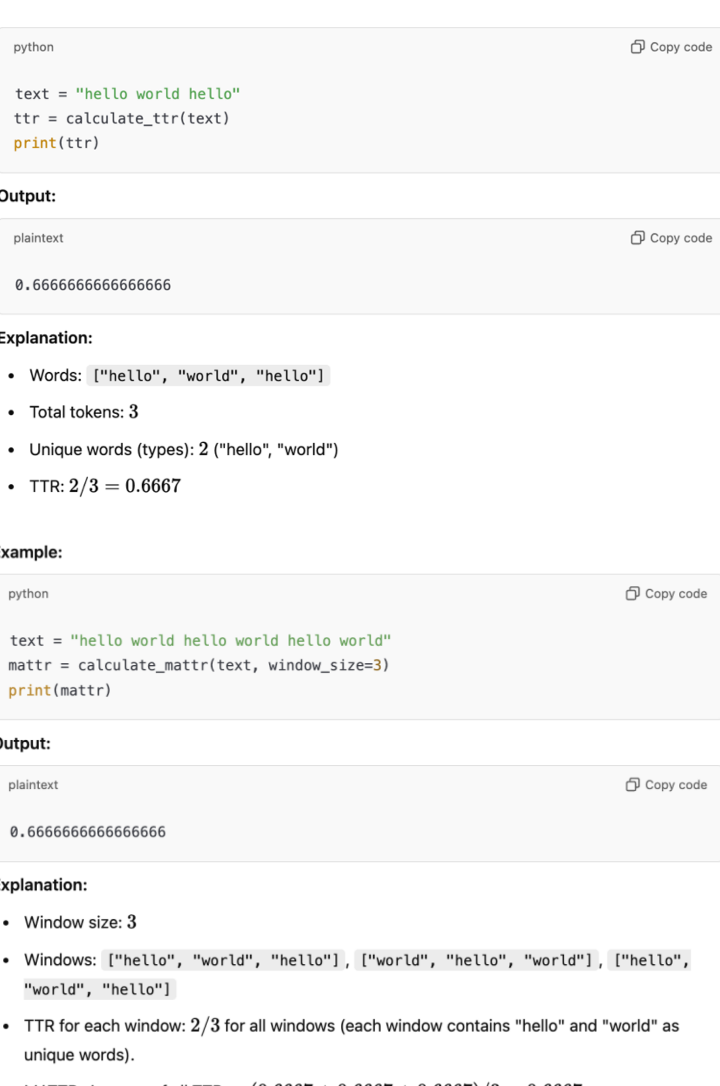
LLM Metrics : Type Token Ratio & Moving-Average Type Token Ratio (4/17)

This function calculates the Type-Token Ratio (TTR), which measures the diversity of a text by comparing the number of unique words (types) to the total number of words (tokens). A higher TTR indicates greater lexical diversity.

The problem is that the TTR of a text sample is affected by its length; obviously, the longer the text goes on, the more likely it is that the next word will be one that has already occurred.

Solution? Moving-Average Type Token Ratio

In that, we choose a window length (say 500 words) and then compute the TTR for words 1–500, then for words 2–501, then 3–502, and so on to the end of the text. The mean of all these TTRs is a measure of the lexical diversity of the entire text and is not affected by text length nor by any statistical assumptions. Further, the individual TTRs can be compared to detect changes within the text. This helps smooth out fluctuations caused by varying text lengths.



LLM Metrics : Average Sentence Length (5/17)

This metric gives the average sentence length of an input text. It is calculated by dividing the total number of words in the text by the total number of sentences. This metric is useful for analysing the complexity of writing style — longer sentences might indicate more complex or formal writing.

Key points:

1. **Shorter sentences** lead to a lower average sentence length, often indicating simple or informal writing.
2. **Longer sentences** lead to a higher average sentence length, often indicating complex or academic writing.

Example:

Output:

plaintext Copy code

```
5.333333333333333
```

Explanation:

1. **Split into sentences:**
   - Sentences: `["Hello world.", "This is a test sentence.", "Let's see how it works."]`
2. **Count total words:**
   - Words in each sentence: `2, 5, 7` (total = 14 words)
3. **Calculate average sentence length:**
   - Average = $\frac{\text{Total Words}}{\text{Number of Sentences}} = \frac{14}{3} \approx 5.33$

LLM Metrics : Stopwords Frequency (6/17)

This is used to calculate the frequency of **function words** (e.g., prepositions, conjunctions, articles, and pronouns) in each text. Function words, often called **stopwords**, are essential for grammatical structure but carry less semantic meaning. The function computes the ratio of function words to the total number of words, helping analyze writing style and formality.

Key Points:

1. **Higher function word frequency**: Often found in formal, dense, or descriptive texts.
2. **Lower function word frequency**: Indicates more content words, typical of creative or informal texts.

Example:

python Copy code

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

# Sample text
text = "The cat sat on the mat and looked at the dog."

# Calculate function word frequency
frequency = calculate_function_word_frequencies(text)
print(frequency)
```

Output:

plaintext Copy code

```plaintext
0.5
```

Explanation:

1. **Tokenize words**:
   - Words: `["the", "cat", "sat", "on", "the", "mat", "and", "looked", "at", "the", "dog"]`
2. **Identify function words**:
   - Function words in the text: `["the", "on", "the", "and", "at", "the"]` (6 total)
3. **Count total words**:
   - Total words: 11
4. **Calculate frequency**:
   - Frequency = $\frac{\text{Function Word Count}}{\text{Total Words}} = \frac{6}{11} \approx 0.5$

LLM Metrics : N-Grams Calculation (7/17)

Bi-grams are consecutive pairs of words, and we identify the top 5 most frequently occurring bi-grams in the text. Tri-grams are consecutive sequences of three words. This analysis helps uncover common word pairs, which can provide insights into writing patterns or repetitive phrases.

Key Points:

1. **Bi-grams** focus on pairs of words and are useful for detecting common phrases or adjacent word usage.
2. **Tri-grams** provide more contextual patterns and are helpful in applications like language modeling and text generation.

Example:

python Copy code

```python
from nltk import word_tokenize
from collections import Counter
import nltk
nltk.download('punkt')

# Sample text
text = "The quick brown fox jumps over the lazy dog. The fox is quick."

# Calculate bi-grams
top_bigrams = calculate_bigrams(text)
print(top_bigrams)
```

Output:

plaintext Copy code

```plaintext
[('the', 'quick'), ('quick', 'brown'), ('brown', 'fox'), ('fox', 'jumps'), ('jumps
```

Explanation:

- Tokenized words: `["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "the", "fox", "is", "quick"]`
- Generated bi-grams: `[("the", "quick"), ("quick", "brown"), ...]`
- Most frequent bi-grams: Top 5 based on frequency.

Example:

python Copy code

```python
# Sample text
text = "The quick brown fox jumps over the lazy dog. The fox is quick."

# Calculate tri-grams
top_trigrams = calculate_trigrams(text)
print(top_trigrams)
```

Output:

plaintext Copy code

```plaintext
[('the', 'quick', 'brown'), ('quick', 'brown', 'fox'), ('brown', 'fox', 'jumps'),
```

Explanation:

- Tokenized words: `["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "the", "fox", "is", "quick"]`
- Generated tri-grams: `[("the", "quick", "brown"), ("quick", "brown", "fox"), ...]`
- Most frequent tri-grams: Top 5 based on frequency.

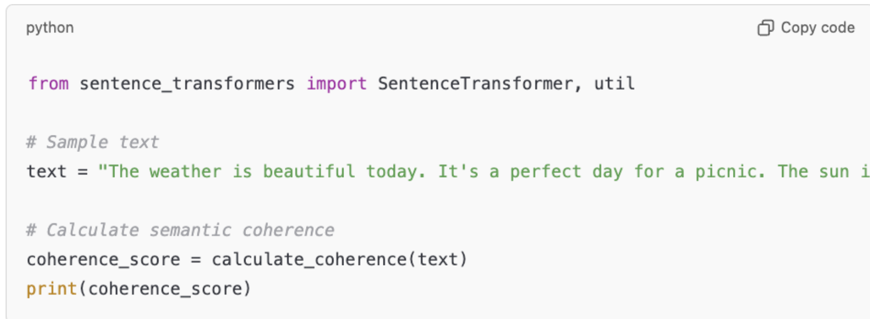
LLM Metrics : Semantic Coherence (8/17)

Semantic coherence of a given text measures how closely related consecutive sentences are. It is done by using embeddings generated from a pre-trained transformer model (in this case, I used Sentence Transformer). Semantic coherence indicates the flow and logical connection between sentences.

Key Points:

1. **High coherence**: Sentences flow logically and are semantically connected.
2. **Low coherence**: Sentences lack logical flow or are unrelated in meaning.

Example:



```python
from sentence_transformers import SentenceTransformer, util

# Sample text
text = "The weather is beautiful today. It's a perfect day for a picnic. The sun is

# Calculate semantic coherence
coherence_score = calculate_coherence(text)
print(coherence_score)
```

Output:

```plaintext
0.85
```

Explanation:

1. **Split sentences**:
   - Sentences: `["The weather is beautiful today.", "It's a perfect day for a picnic.", "The sun is shining brightly."]`
2. **Generate sentence embeddings**:
   - Each sentence is converted into a numerical representation (embedding) using the `SentenceTransformer` model.
3. **Calculate cosine similarity**:
   - Cosine similarity is calculated for consecutive sentence embeddings to measure their semantic relatedness.
   - Example: Similarity between "The weather is beautiful today." and "It's a perfect day for a picnic."
4. **Compute average coherence**:
   - Average the similarities to determine the overall semantic coherence of the text.

LLM Metrics : POS Tagging (9/17)

Part-of-Speech (POS) tagging on a given text categorizes the counts of different POS tags into predefined categories (e.g., nouns, verbs, modifiers). POS tagging identifies the grammatical role of each word in the text (like noun, verb, adjective), and categorizing these tags helps in understanding the structure and style of the text.

Key Points:

1. **POS tagging** reveals the grammatical structure of the text.
2. **Categorizing POS tags** helps in studying patterns, such as the use of descriptive modifiers, action verbs, or formal nouns.
3. Useful for tasks like **style analysis**, **genre classification**, or **text complexity evaluation**.

Example:

```python
import spacy
import pandas as pd

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

# Example text
text = "The quick brown fox jumps over the lazy dog."

# Calculate POS counts
pos_counts = categorize_pos_counts(text)
print(pos_counts)
```

Output:

```plaintext
{
    'Nouns': 3,
    'Verbs': 1,
    'Modifiers': 2,
    'Pronouns': 0,
    'Determiners_Particles': 1,
    'Conjunctions': 0,
    'Adpositions': 1,
    'Punctuation_Symbols': 1,
    'Spaces': 0
}
```

Explanation:

1. **Tokenize text:**
   - Words: `["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]`
2. **Identify POS tags:**
   - For example, "fox" is a noun, "jumps" is a verb, "quick" is an adjective.
3. **Categorize POS counts:**
   - Nouns: 3 ("fox", "dog", "brown")
   - Verbs: 1 ("jumps")
   - Modifiers: 2 ("quick", "lazy")
   - Other categories as per the tagging.

LLM Metrics : Word Repetition Analysis (10/17)

I performed word repetition analysis to find **word repetitions** in each text. It identifies words that occur more than once and calculates the **repetition ratio**, which is the proportion of repeated word occurrences to the total number of words. This helps in understanding the redundancy or emphasis in the text.

Key Points:

1. **Repeating words**: Indicates which words are repeated, potentially showing emphasis or redundancy.
2. **Repetition ratio**: Quantifies the extent of repetition in the text.

This function is useful for analyzing text styles, identifying redundancy in writing, or detecting emphasis in speech transcripts or creative writing.

Example:

python Copy code

```python
from nltk.tokenize import word_tokenize
from collections import Counter
import nltk
nltk.download('punkt')

# Sample text
text = "The quick brown fox jumps over the lazy dog. The fox is quick and very qui

# Perform word repetition analysis
repeating_words, repetition_ratio = word_repetition_analysis(text)
print("Repeating Words:", repeating_words)
print("Repetition Ratio:", repetition_ratio)
```

Output:

plaintext Copy code

```
Repeating Words: {'the': 3, 'quick': 3, 'fox': 2}
Repetition Ratio: 0.5
```

Explanation:

1. **Tokenize the text:**
   - Words: `["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "the", "fox", "is", "quick", "and", "very", "quick"]`
2. **Count word frequencies:**
   - Word counts: `{'the': 3, 'quick': 3, 'fox': 2, ...}`
3. **Identify repeating words:**
   - Words occurring more than once: `{'the': 3, 'quick': 3, 'fox': 2}`
4. **Calculate repetition ratio:**
   - Total words: 16
   - Repeated occurrences: $3 + 3 + 2 = 8$
   - Ratio: $8/16 = 0.5$

LLM Metrics : Readability Score (11/17)

In this function, I calculate the Flesch-Kincaid readability score of a given text. The score measures how easy a text is to read, based on the average number of words per sentence and syllables per word. A higher score indicates easier readability, while a lower score suggests the text is more complex.

Key Points:

1. **Higher score:** Easier to read (e.g., children's books or simple instructions).
2. **Lower score:** More complex text (e.g., academic papers or legal documents).

Example:

```python
from textblob import TextBlob
import syllapy

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

# Calculate Flesch-Kincaid readability score
fk_score = flesch_kincaid(text)
print("Flesch-Kincaid Score:", fk_score)
```

Output:

```plaintext
Flesch-Kincaid Score: 104.1
```

Explanation:

1. **Text analysis:**
   - Sentences: `["The quick brown fox jumps over the lazy dog.", "It is a sunny day."]`
   - Words: `["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "It", "is", "a", "sunny", "day"]`
   - Number of sentences: 2
   - Number of words: 14
   - Number of syllables: 18 (e.g., "quick" = 1 syllable, "brown" = 1 syllable, "sunny" = 2 syllables)
2. **Flesch-Kincaid formula:**

$$\text{FK Score} = 206.835 - 1.015 \cdot (\text{Average Words per Sentence}) - 84.6 \cdot (\text{Average Syllables per Word})$$

   - Average words per sentence: $14/2 = 7$
   - Average syllables per word: $18/14 \approx 1.29$
   - Score: $206.835 - 1.015 \cdot 7 - 84.6 \cdot 1.29 \approx 104.1$

LLM Metrics : Sentiment Polarity and Subjectivity (12/17)

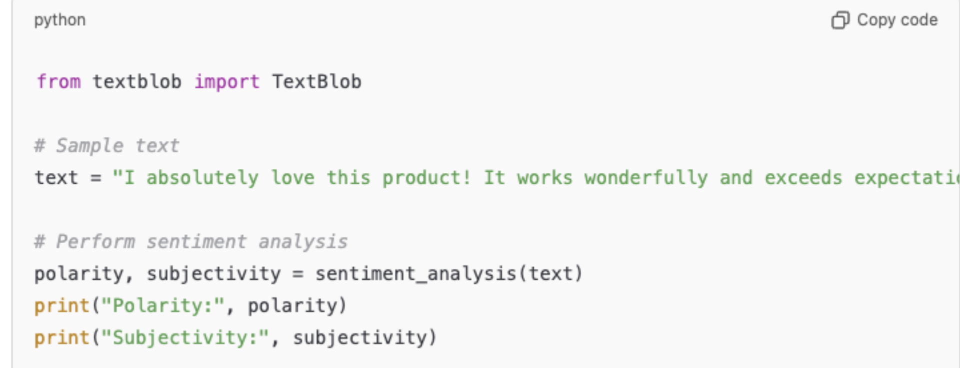
In this function, I perform sentiment analysis on a given text. I calculated:

- **Polarity**: A value between -1 and 1 that indicates the sentiment of the text. Negative values represent negative sentiment, positive values represent positive sentiment, and 0 represents neutral sentiment.
- **Subjectivity**: A value between 0 and 1 that indicates how subjective or opinionated the text is. Higher values represent more subjective or personal opinions, while lower values represent more factual content.

Key Points:

1. **Polarity** identifies the emotional tone (positive, neutral, or negative).
2. **Subjectivity** measures how opinionated or fact-based the content is.

Example:



```python
from textblob import TextBlob

# Sample text
text = "I absolutely love this product! It works wonderfully and exceeds expectati

# Perform sentiment analysis
polarity, subjectivity = sentiment_analysis(text)
print("Polarity:", polarity)
print("Subjectivity:", subjectivity)
```

Output:

plaintext Copy code

```
Polarity: 0.9
Subjectivity: 1.0
```

Explanation:

1. **Polarity**:
   - The text is overwhelmingly positive, with phrases like "absolutely love" and "exceeds expectations," resulting in a high polarity score of 0.9.
2. **Subjectivity**:
   - The text reflects personal opinions and feelings, making it highly subjective with a subjectivity score of 1.0.

LLM Metrics : Interrogative Content (13/17)

In this function, I analyze the interrogative content of a given text by identifying and counting the number of questions. It uses two criteria to detect questions:

1. Sentences ending with a **question mark** (?).
2. Sentences that start with common **question words** or subject-auxiliary inversion patterns (e.g., "What," "Why," "Is," "Can").

Key Points:

1. Questions often indicate inquiry or engagement in a text.
2. The function is robust, detecting questions even if the text lacks a question mark but follows typical interrogative patterns.

Example:

```python
import re
from nltk.tokenize import sent_tokenize
import pandas as pd

# Sample text
text = "What is your name? I like this product. Can you help me? Why is this happe

# Count the number of questions
num_questions = count_questions(text)
print("Number of Questions:", num_questions)
```

Output:

Explanation:

1. **Split into sentences**:
   - Sentences: `["What is your name?", "I like this product.", "Can you help me?", "Why is this happening?"]`
2. **Identify questions**:
   - Sentence 1: Ends with "?" → Question
   - Sentence 2: Does not meet criteria → Not a question
   - Sentence 3: Starts with "Can" and ends with "?" → Question
   - Sentence 4: Starts with "Why" and ends with "?" → Question
3. **Count questions**:
   - Total questions: 3

LLM Metrics : Cognitive Verbs (14/17)

In this function, I count the occurrences of cognitive verbs in each text. Cognitive verbs are action words associated with mental processes like thinking, analysing, evaluating, or creating. These verbs are often indicators of higher-order cognitive activity and are useful for assessing the cognitive load or complexity of the text.

Key Points:

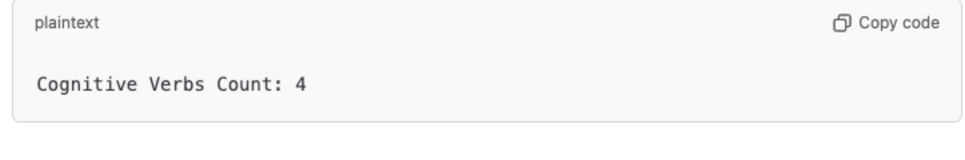
1. **Cognitive verbs**: Words like "analyze," "compare," and "summarize" reflect critical thinking or problem-solving.
2. **Higher count**: Indicates texts that demand intellectual effort, like instructions, academic papers, or problem statements.
3. **Applications**: Useful for assessing instructional materials, academic writing, or evaluating the complexity of tasks in a text

Example:

python Copy code

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

# Sample text
text = "The student needs to analyze the data and compare the results. Then, they :

# Count cognitive verbs
cognitive_count = cognitive_verbs_count(text)
print("Cognitive Verbs Count:", cognitive_count)
```

Output:



Explanation:

1. **Tokenize the text:**
   - Words: `["the", "student", "needs", "to", "analyze", "the", "data", "and", "compare", "the", "results", "then", "they", "should", "summarize", "their", "findings", "and", "propose", "solutions"]`
2. **Identify cognitive verbs:**
   - Verbs: `["analyze", "compare", "summarize", "propose"]`
3. **Count occurrences:**
   - Total cognitive verbs: 4

LLM Metrics : Special Characters (15/17)

In this function, I calculate the number of special characters in each text. Special characters include symbols like @, #, $, %, ^, &, *, (,), _, +, =, and -. These characters are often used in technical documents, code snippets, or casual text (like social media posts).

Key Points:

1. **Special characters** often appear in:
   1. **Emails** (e.g., @)
   2. **Hashtags** or **handles** (e.g., #)
   3. **Currency values** (e.g., $)
   4. **Equations or programming** (e.g., +, -, *, /)
2. **Higher count**: May indicate technical, informal, or casual text.
3. **Applications**: Useful for text classification, detecting technical content, or filtering out noisy text.

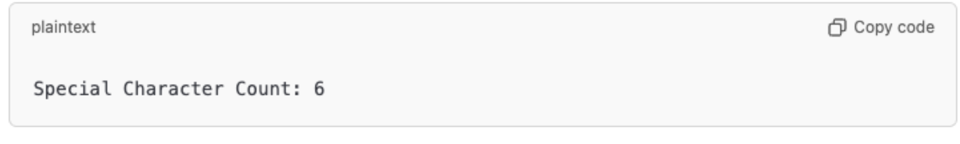
Example:

python

Copy code

```python
import re

# Sample text
text = "Email me at hello@example.com or call #12345 for details! Cost: $100."

# Calculate special character count
special_char_count = character_level_features(text)
print("Special Character Count:", special_char_count)
```

Output:



Explanation:

1. **Identify special characters:**
   - Special characters in the text: `["@", "#", "$", "(", ")", "-"]`
2. **Count occurrences:**
   - Total special characters: 6

LLM Metrics : Spelling Errors (16/17)

In this function, I identify and count the number of spelling errors in each text. It compares each word in the text against a dictionary of correctly spelled words. Words not found in the dictionary are considered misspelled. This helps evaluate the grammatical quality of the text.

Key Points:

1. **Spelling errors** often indicate informal writing, typos, or low text quality.
2. **Higher error count**: Indicates poor grammar or carelessness.

Example:

python Copy code

```python
from spellchecker import SpellChecker
import re

# Sample text
text = "The qwick brown fox jmps over the lazi dog."

# Calculate spelling errors
error_count = spelling_errors(text)
print("Number of Spelling Errors:", error_count)
```

Output:

plaintext Copy code

```
Number of Spelling Errors: 5
```

Explanation:

1. **Tokenize and preprocess text:**
   - Words: `["the", "qwick", "brown", "fox", "jmps", "over", "the", "lazi", "dog"]`
2. **Check against dictionary:**
   - Misspelled words: `["qwick", "jmps", "lazi"]`
3. **Count errors:**
   - Total errors: 3

LLM Metrics : Grammar Errors (17/17)

In this function, I count the number of grammatical errors in each text using the LanguageTool library. It scans the text for grammar issues, such as incorrect verb tense, subject-verb agreement errors, or improper punctuation, and returns the total number of detected errors.

Key Points:

1. Captures a variety of issues, such as tense mismatches, punctuation errors, or spelling mistakes in context.
2. Used for Evaluating the grammatical correctness of datasets for NLP tasks.
3. Useful for both formal and casual writing to ensure clarity and correctness.

Example:

python Copy code

```python
import language_tool_python
import pandas as pd

# Initialize LanguageTool
tool = language_tool_python.LanguageTool('en-US')

# Sample text
text = "The cat climb the tree quickly. She dont like the rain."

# Calculate grammar errors
error_count = grammar_errors_count(text)
print("Number of Grammar Errors:", error_count)
```

Output:

Explanation:

1. **Analyze the text:**
   - Text: `"The cat climb the tree quickly. She dont like the rain."`
2. **Identify grammar issues:**
   - Error 1: `"climb"` should be `"climbed"` (incorrect verb tense).
   - Error 2: `"dont"` should be `"doesn't"` (missing apostrophe).
3. **Count errors:**
   - Total errors: 2

Feature Correlation

1. The heatmap reveals significant correlations among linguistic features, particularly within groups like grammatical components (e.g., Pronouns, Verbs, Determiners, etc.), where values exceed 0.7.

2. Perplexity and Character Entropy exhibit relatively weak correlations with most other features, suggesting they capture distinct aspects of the data.

3. Certain feature clusters, such as those related to sentence structure (Sentence Length, Stopwords Frequency) and grammatical categories, show cohesive patterns, indicating logical grouping based on shared linguistic functions.

Correlation Heatmap

	Perplexity	CharacterEntropy	WordEntropy	Burstiness	TTR	MATTR	ŠentenceLength	StopwordFrequency	SemanticCoherence	Nouns	Verbs	Modifiers	Pronouns	Determiners_Particles	Conjunctions	Adpositions	Punctuation_Symbols	Spaces	Repetition Ratio	Repeating N-grams Count	Sentiment Polarity	Sentiment Subjectivity	Readability Score	Num_Questions	Cognitive_Verbs	Special_Char	Spelling_Errors	Grammar_Errors
Perplexity	1.00	-0.11	-0.27	-0.20	0.39	0.04	0.08	-0.01	-0.05	-0.29	-0.33	-0.31	-0.25	-0.27	-0.31	-0.29	-0.35	-0.24	-0.43	-0.37	-0.01	-0.00	-0.10	-0.11	-0.18	-0.11	0.04	0.08
CharacterEntropy	-0.11	1.00	0.31	-0.15	0.23	0.32	-0.25	-0.33	-0.17	0.10	-0.01	0.07	-0.02	-0.03	-0.05	0.05	0.35	0.22	-0.11	-0.02	-0.01	-0.03	0.14	0.18	-0.01	0.23	0.13	-0.11
WordEntropy	-0.27	0.31	1.00	0.15	0.03	0.54	-0.09	-0.28	-0.24	0.70	0.59	0.69	0.34	0.56	0.49	0.66	0.67	0.29	-0.00	0.03	-0.16	-0.12	-0.00	0.19	0.24	0.30	0.37	0.19
Burstiness	-0.20	-0.15	0.15	1.00	-0.61	-0.43	0.11	0.43	-0.02	0.31	0.57	0.36	0.54	0.47	0.52	0.36	0.21	0.17	0.51	0.32	0.03	-0.01	0.21	0.11	0.13	0.00	0.15	0.32
TTR	0.39	0.23	0.03	-0.61	1.00	0.66	-0.15	-0.50	-0.17	-0.39	-0.65	-0.43	-0.60	-0.56	-0.64	-0.45	-0.29	-0.24	-0.94	-0.76	-0.10	-0.03	-0.17	-0.05	-0.24	-0.00	-0.05	-0.23
MATTR	0.04	0.32	0.54	-0.43	0.66	1.00	-0.20	-0.63	-0.12	0.08	-0.16	0.10	-0.29	-0.18	-0.20	0.02	0.16	-0.02	-0.57	-0.35	-0.15	-0.08	-0.19	0.04	-0.00	0.14	0.01	-0.22
ŠentenceLength	0.08	-0.25	-0.09	0.11	-0.15	-0.20	1.00	0.20	-0.03	0.03	0.08	0.03	0.10	0.10	0.14	0.06	-0.15	-0.06	0.06	0.01	-0.00	-0.00	-0.64	-0.11	0.02	-0.01	0.06	0.18
StopwordFrequency	-0.01	-0.33	-0.28	0.43	-0.50	-0.63	0.20	1.00	-0.15	-0.20	0.27	-0.07	0.52	0.23	0.30	-0.00	-0.25	-0.01	0.45	0.11	0.12	0.14	0.39	0.07	-0.09	-0.25	-0.01	0.26
SemanticCoherence	-0.05	-0.17	-0.24	-0.02	-0.17	-0.12	-0.03	-0.15	1.00	0.06	-0.08	-0.01	-0.21	-0.06	-0.03	0.03	-0.08	-0.14	0.10	0.22	0.04	0.02	-0.25	-0.19	0.16	0.01	-0.26	-0.25
Nouns	-0.29	0.10	0.70	0.31	-0.39	0.08	0.03	-0.20	0.06	1.00	0.69	0.74	0.29	0.78	0.62	0.88	0.71	0.32	0.30	0.41	-0.09	-0.15	-0.17	0.02	0.41	0.41	0.40	0.26
Verbs	-0.33	-0.01	0.59	0.57	-0.65	-0.16	0.08	0.27	-0.08	0.69	1.00	0.77	0.80	0.83	0.87	0.74	0.63	0.32	0.59	0.52	-0.01	-0.04	0.17	0.19	0.32	0.17	0.28	0.35
Modifiers	-0.31	0.07	0.69	0.36	-0.43	0.10	0.03	-0.07	-0.01	0.74	0.77	1.00	0.53	0.70	0.72	0.77	0.70	0.29	0.39	0.44	-0.03	-0.04	-0.05	0.14	0.30	0.23	0.24	0.21
Pronouns	-0.25	-0.02	0.34	0.54	-0.60	-0.29	0.10	0.52	-0.21	0.29	0.80	0.53	1.00	0.56	0.75	0.45	0.42	0.27	0.59	0.43	0.08	0.07	0.34	0.26	0.14	-0.00	0.16	0.35
Determiners_Particles	-0.27	-0.03	0.56	0.47	-0.56	-0.18	0.10	0.23	-0.06	0.78	0.83	0.70	0.56	1.00	0.71	0.78	0.60	0.31	0.47	0.43	-0.04	-0.03	0.07	0.14	0.34	0.22	0.36	0.34
Conjunctions	-0.31	-0.05	0.49	0.52	-0.64	-0.20	0.14	0.30	-0.03	0.62	0.87	0.72	0.75	0.71	1.00	0.65	0.55	0.27	0.58	0.52	-0.02	-0.01	0.11	0.16	0.30	0.14	0.23	0.32
Adpositions	-0.29	0.05	0.66	0.36	-0.45	0.02	0.06	-0.00	0.03	0.88	0.74	0.77	0.45	0.78	0.65	1.00	0.68	0.30	0.36	0.43	-0.06	-0.10	-0.08	0.06	0.34	0.30	0.31	0.22
Punctuation_Symbols	-0.35	0.35	0.67	0.21	-0.29	0.16	-0.15	-0.25	-0.08	0.71	0.63	0.70	0.42	0.60	0.55	0.68	1.00	0.37	0.36	0.47	-0.05	-0.04	0.10	0.29	0.29	0.43	0.29	0.09
Spaces	-0.24	0.22	0.29	0.17	-0.24	-0.02	-0.06	-0.01	-0.14	0.32	0.32	0.29	0.27	0.31	0.27	0.30	0.37	1.00	0.27	0.25	0.03	-0.00	0.13	0.16	0.15	0.18	0.10	0.15
Repetition Ratio	-0.43	-0.11	-0.00	0.51	-0.94	-0.57	0.06	0.45	0.10	0.30	0.59	0.39	0.59	0.47	0.58	0.36	0.36	0.27	1.00	0.74	0.09	0.05	0.29	0.15	0.18	-0.00	0.03	0.18
Repeating N-grams Count	-0.37	-0.02	0.03	0.32	-0.76	-0.35	0.01	0.11	0.22	0.41	0.52	0.44	0.43	0.43	0.52	0.43	0.47	0.25	0.74	1.00	0.07	0.04	0.05	0.05	0.27	0.11	-0.01	0.02
Sentiment Polarity	-0.01	-0.01	-0.16	0.03	-0.10	-0.15	-0.00	0.12	0.04	-0.09	-0.01	-0.03	0.08	-0.04	-0.02	-0.06	-0.05	0.03	0.09	0.07	1.00	0.29	0.04	-0.04	-0.01	-0.04	-0.11	-0.05
Sentiment Subjectivity	-0.00	-0.03	-0.12	-0.01	-0.03	-0.08	-0.00	0.14	0.02	-0.15	-0.04	-0.04	0.07	-0.03	-0.01	-0.10	-0.04	-0.00	0.05	0.04	0.29	1.00	0.04	0.05	0.01	-0.06	-0.09	-0.07
Readability Score	-0.10	0.14	-0.00	0.21	-0.17	-0.19	-0.64	0.39	-0.25	-0.17	0.17	-0.05	0.34	0.07	0.11	-0.08	0.10	0.13	0.29	0.05	0.04	0.04	1.00	0.29	-0.15	-0.14	0.09	0.14
Num_Questions	-0.11	0.18	0.19	0.11	-0.05	0.04	-0.11	0.07	-0.19	0.02	0.19	0.14	0.26	0.14	0.16	0.06	0.29	0.16	0.15	0.05	-0.04	0.05	0.29	1.00	-0.05	-0.02	0.13	0.11
Cognitive_Verbs	-0.18	-0.01	0.24	0.13	-0.24	-0.00	0.02	-0.09	0.16	0.41	0.32	0.30	0.14	0.34	0.30	0.34	0.29	0.15	0.18	0.27	-0.01	0.01	-0.15	-0.05	1.00	0.17	-0.02	-0.05
Special_Char	-0.11	0.23	0.30	0.00	-0.00	0.14	-0.01	-0.25	0.01	0.41	0.17	0.23	-0.00	0.22	0.14	0.30	0.43	0.18	-0.00	0.11	-0.04	-0.06	-0.14	-0.02	0.17	1.00	0.21	0.02
Spelling_Errors	0.04	0.13	0.37	0.15	-0.05	0.01	0.06	-0.01	-0.26	0.40	0.28	0.24	0.16	0.36	0.23	0.31	0.29	0.10	0.03	-0.01	-0.11	-0.09	0.09	0.13	-0.02	0.21	1.00	0.75
Grammar_Errors	0.08	-0.11	0.19	0.32	-0.23	-0.22	0.18	0.26	-0.25	0.26	0.35	0.21	0.35	0.34	0.32	0.22	0.09	0.15	0.18	0.02	-0.05	-0.07	0.14	0.11	-0.05	0.02	0.75	1.00

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

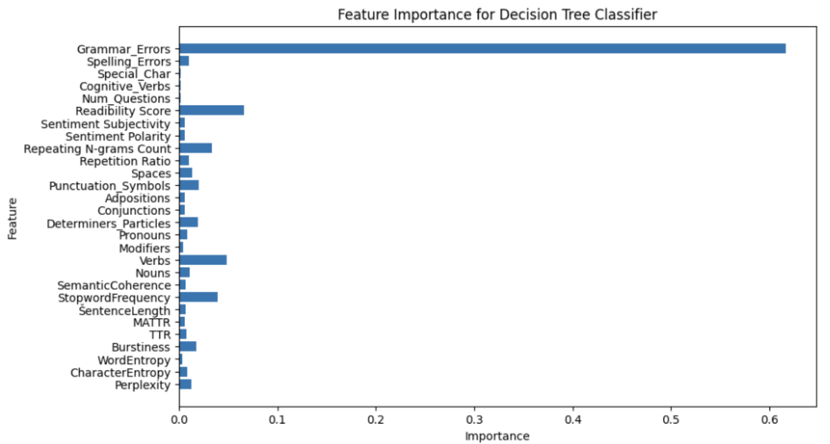
Results (DAIGT | Catch The AI)

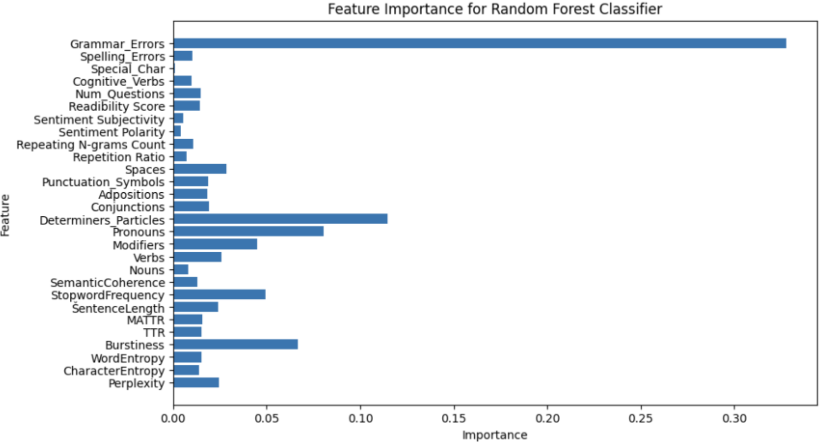
Validation

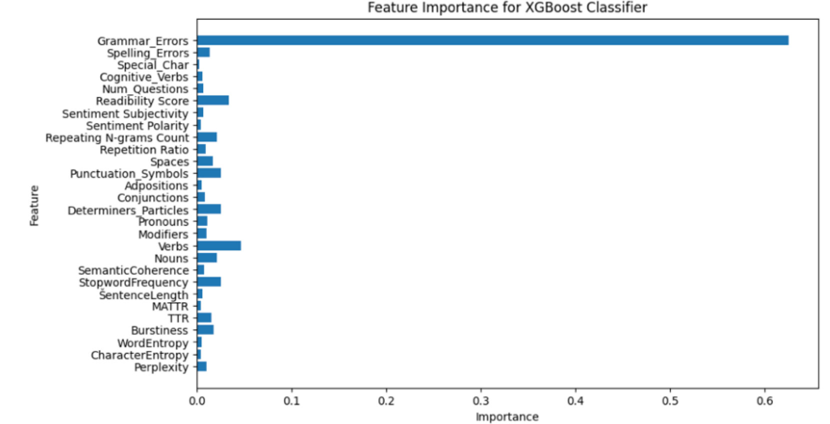
Classification Algorithms	Accuracy	F1-score
Logistic Regression	0.9344	0.9353
K-Nearest Neighbor	0.9322	0.9335
SVM (Linear)	0.9516	0.9523
SVM (Polynomial)	0.9766	0.9766
SVM (Gaussian)	0.9722	0.9722
Naïve Bayes Classifier	0.8300	0.8426
Decision Tree	0.9498	0.9502
Random Forest	0.9326	0.9334
XGBoost	0.9806	0.9806
Multilayer Perceptron	0.9813	0.9813

Testing

Classification Algorithms	Accuracy	F1-score
Multilayer Perceptron	0.9758	0.9758
BERT (for baseline)	0.9765	0.9770







Results (DAIGT - Mixed Paragraph Dataset v1)

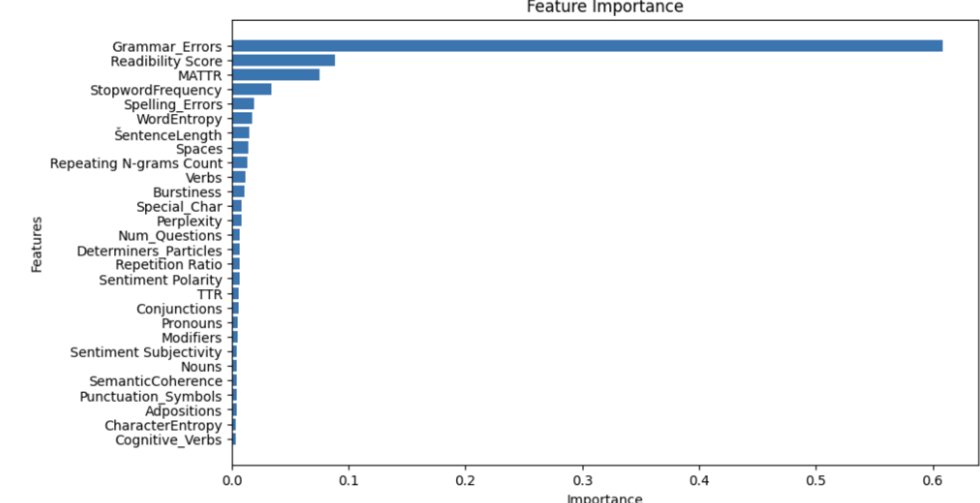
	Classification Algorithms	Accuracy	F1-score
Validation	Logistic Regression	0.8449	0.8446
	K-Nearest Neighbor	0.7860	0.7863
	SVM (Linear)	0.8540	0.8538
	SVM (Polynomial)	0.8676	0.8674
	SVM (Gaussian)	0.8617	0.8615
	Naïve Bayes Classifier	0.7003	0.6965
	Decision Tree	0.8023	0.8023
	Random Forest	0.8088	0.8058
	XGBoost	0.8738	0.8736
	Multilayer Perceptron	0.8710	0.8698
Testing	XGBoost	0.8669	0.8665

```
The best model is: XGBoost Classifier with a validation accuracy of 0.8738

Feature Importance:
                   Feature  Importance
27          Grammar_Errors    0.421843
22       Readibility Score    0.041550
10                   Verbs    0.038701
16     Punctuation_Symbols    0.038278
7        StopwordFrequency    0.038030
4                      TTR    0.037842
17                  Spaces    0.033999
18        Repetition Ratio    0.033853
19  Repeating N-grams Count   0.033188
26         Spelling_Errors    0.028001
9                    Nouns    0.020656
13   Determiners_Particles    0.019307
12                Pronouns    0.018120
15             Adpositions    0.017254
3               Burstiness    0.016847
6           ŠentenceLength    0.014980
24         Cognitive_Verbs    0.013952
8        SemanticCoherence    0.013658
14            Conjunctions    0.013603
23           Num_Questions    0.012856
11               Modifiers    0.012786
0               Perplexity    0.012628
5                    MATTR    0.012392
1         CharacterEntropy    0.012280
2              WordEntropy    0.011649
25            Special_Char    0.011301
21  Sentiment Subjectivity    0.010730
20      Sentiment Polarity    0.009714
```

Results (LLM - Detect AI Generated Text Dataset)

	Classification Algorithms	Accuracy	F1-score
Validation	Logistic Regression	0.9415	0.9415
	K-Nearest Neighbor	0.9242	0.9243
	SVM (Linear)	0.9602	0.9603
	SVM (Polynomial)	0.9744	0.9745
	SVM (Gaussian)	0.9650	0.9650
	Naïve Bayes Classifier	0.8840	0.8847
	Decision Tree	0.9689	0.9690
	Random Forest	0.9538	0.9536
	XGBoost	0.9885	0.9885
	Multilayer Perceptron	0.9832	0.9832
Testing	XGBoost	0.9871	0.9871

```
The best model is: XGBoost Classifier with a validation accuracy of 0.9885

Feature Importance:
                   Feature  Importance
27          Grammar_Errors    0.608644
22       Readibility Score    0.088213
5                    MATTR    0.075044
7        StopwordFrequency    0.033593
26         Spelling_Errors    0.018995
2              WordEntropy    0.017207
6           ŠentenceLength    0.014661
17                  Spaces    0.013730
19  Repeating N-grams Count   0.013367
10                   Verbs    0.011560
3               Burstiness    0.010628
25            Special_Char    0.008654
0               Perplexity    0.008555
23           Num_Questions    0.007012
13    Determiners_Particles   0.006596
18        Repetition Ratio    0.006543
20      Sentiment Polarity    0.006281
4                      TTR    0.005964
14            Conjunctions    0.005814
12                Pronouns    0.005387
11               Modifiers    0.005329
21  Sentiment Subjectivity    0.004344
9                    Nouns    0.004275
8        SemanticCoherence    0.004250
16      Punctuation_Symbols   0.004231
15             Adpositions    0.004227
1         CharacterEntropy    0.003533
24          Cognitive_Verbs   0.003365
```



Results (Website): Shows AI generated text to any grammatically correct input text. Input text was copied from Wikipedia.

Results (Website): Shows Likely not AI only when we have grammatical errors in the provided input text

Challenges

1. Working with large language models (LLMs) require significant computational resources, including high-performance GPUs. Training large models can take hours/days, depending on the dataset size and model complexity, causing delays. Limited availability of advanced hardware made it challenging for me to prepare the derived features and run BERT for baseline.
2. To achieve fair and robust performance, the dataset used for training or evaluation must accurately represent the diversity and generality of real-world scenarios. A biased or unrepresentative dataset can lead to models that fail to generalize across different use cases or domains. Example. Kaggle Dataset 1, 2 and 3 had grammatical errors as a major deciding factor when it comes for human written text vs LLM generated text. In real-world, if a human is well versed with English language and makes no grammatical errors, his/her text will be marked as generated by AI.
3. Differentiating between AI-generated and human-generated text is an emerging problem with limited research. Therefore, it is challenging to develop benchmarks or metrics for reliably distinguishing them.

Future Scope

1. Enhance the analysis by using multiple variations of the Type-Token Ratio (TTR) to gain deeper insights into lexical diversity. For example: Root TTR, Corrected TTR etc. These variations provide complementary perspectives on lexical diversity, making the analysis more comprehensive and adaptable to different text types or lengths.

2. Expand the evaluation framework by incorporating additional readability metrics to capture the complexity of text from various angles. For example: Coleman-Liau Index, Automated Readability Index (ARI), SMOG Index (Simple Measure of Gobbledygook) etc. By using multiple algorithms, you can offer a more nuanced evaluation of text readability and adapt the analysis to different target audiences or domains.

3. Test the robustness, scalability, and generalizability of the methodology by applying it to a larger dataset. A larger dataset provides a "big-picture" view of my methodology, revealing potential limitations, edge cases, or areas for improvement. Recently found dataset : Human vs. LLM Text Corpus consisting of 788922 unique records. (Link)

References

1. GitHub Repository : LLMMetricResearch

(Currently a private repository. Will make it public after submitting the project)