Pipeline Version 2

March 26, 2024

Contents

Executive Summary

In this second pipeline, I aimed to build on the shortcomings of our first pipeline by increasing our data through text augmentation and experimenting with newer models to improve our model's performance.

For text augmentation, I experimented with an LSTM model to generate text that mimicked my writing style. Due to poor results, I fine-tuned a DistilGPT-2 model to achieve better outcomes. However, these text generation attempts were unsuccessful in providing satisfactory augmented data. Consequently, I resorted to using external models for text augmentation, utilizing techniques such as paraphrasing and back translation to augment our text data.

With a substantially larger dataset, we selected Extremely Randomized Trees and XGBoost to train on our classification problem on differentiating between ChatGPT and human text. The results still showcased overfitting, and we performed rigorous performance comparisons in a systematic manner to validate if my suspicions of overfitting were valid.

Section 1: Introduction

With the rise in capabilities of chatGPT, there has been a lot of conversation about its efficacy and when and where it is right to use it. Conversely, most writers have claimed that it is unable to replicate each writer's uniqueness. Well, considering that chatGPT is trained on a large corpus of human text, I got pretty curious about whether there really is such a thing as uniqueness in human writing. Though openAI recently pulled down their AI text detector, I decided to explore the possibility of being able to distinguish between human and AI-generated text for the following reasons:

- Most chatGPT responses have a relatively consistent structure when prompted to answer questions without explicit instruction about the answer format in the prompt.
- Most humans, to some degree, can distinguish between AI and human text; thus, if humans can do it, then we can definitely, though not perfectly,find ways to replicate the patterns they use it.

First Pipeline results

In the first pipeline, I adopted a simple-to-complex model selection philosophy, which led me to select logistic regression as the first model to throw at the aforementioned problem in the introduction.

	precision	recall	f1-score	support
0	0.95	0.92	0.94	609
1	0.90	0.94	0.92	449
accuracy			0.93	1058
macro avg	0.92	0.93	0.93	1058
weighted avg	0.93	0.93	0.93	1058

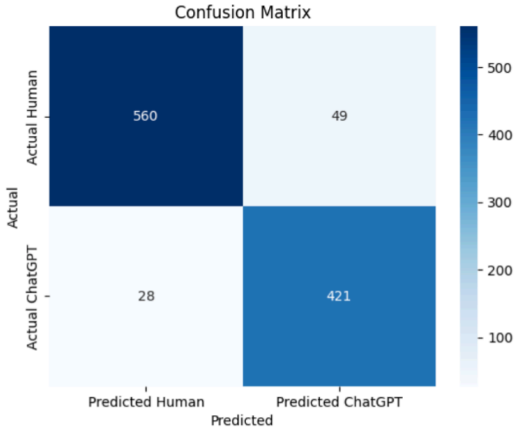


Figure 1

Classification Report and Confusion Matrix detailing precision, recall, and F1-scores for human versus ChatGPT text identification, with the matrix revealing the number of true and false predictions. As one can observe, the precision, F1, and recall scores are extremely high.

From the image above, we can see that the model performed well, and it is quite apparent that the the model may have overfit. In fact, this suspicion of overfitting is proven to be true when the model is presented with data it had never seen before, and the results are pretty different.

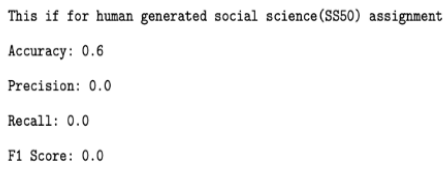


Figure 2. This showcases the accuracy score of the model when presented with samples of some SS50 assignments.

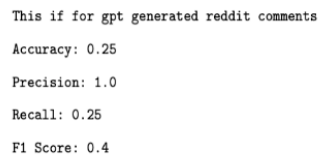


Figure 3. This showcases the accuracy score for the model when presented with external data from GTP generated reddit comments.

Additionally, a comparative performance analysis with a more sophisticated model, such as Bidirectional Encoder Representations from Transformers (BERT), yielded similar outcomes.

So why did our model overfit ?

As stated in the discussion of the first pipeline as well as the received feedback, the biggest factor that led to the model overfitting was simply the lack of sufficient data. To be specific, when working with small datasets the models tend to get artificially "high" performance because the small dataset acts as an under-constraining set of examples, failing to provide a diverse and comprehensive representation of the real-world variability.

So how do I plan to resolve this ?

Ironically, the goal of this assignment would be to have the model perform worse, as it can be an indicator that the model is not overfitting. To address this, I plan to use an external chatGPT response dataset to get more GPT-generated data. I selected this dataset due to the variety of responses it provides with the hope of having a more robust model. The selected dataset can be found here.

However, in order to preserve the integrity of my writing style, I decided to augment my own text rather than using an external dataset. To do this, I took two approaches: - Fine-tuning an already pre-existing model to mimic my writing style. - Leveraging text augmentation techniques like paraphrasing and context embedding to enrich and diversify my dataset. The specifics of these approaches are discussed later in this paper.

Secondly, to still address this overfitting issue, I shift towards a 50/50 split for cross-validation to ensure that the model's ability to generalize is rigorously tested. As we can observe from the analysis of the first pipeline, the model did very poorly when presented with external data, and I hope this is a step in the right direction. Also, I do not see this posing an issue now that we have more data due to the augmentation strategies mentioned above.

Finally, we plan to throw new models to this classification problem to assess if we can get better performance.

Data Collection

I developed a Python script to segment text from a Google Doc into manageable chunks, subsequently organizing them into a CSV file structured with columns for Text, Source, Origin (human or ChatGPT), Length (measured in characters), and Topic. My approach to data collection involved revisiting all the text I had produced before the advent of ChatGPT. This corpus encompassed everything from college application essays and assignments to emails and Telegram conversations with friends. I consolidated these texts into a singular Google document, which was then processed by my script to create analyzable segments, each appropriately labeled.

For the text generated by ChatGPT, I meticulously reviewed my past interactions with ChatGPT, extracting the model's responses and compiling them into another Google document. This document was similarly processed through my script for chunking. It's critical to highlight that the majority of these ChatGPT responses fall within the academic realms of computer science, economics, and finance, potentially introducing a certain degree of bias into the training and testing data

Sampling Approach

In gathering my personal text data, I chose a methodical approach over random sampling. My process involved systematically collecting texts from Google Drive, starting from the earliest backup available and concluding just before the introduction of ChatGPT. This deliberate strategy was chosen to safeguard my dataset from potential contamination by inadvertently including text segments derived from GPT. In contrast, for sourcing ChatGPT responses, I adopted a random selection strategy to ensure a diverse and unbiased sample.

Text Chunking process

To ensure the segments we add to our CSV are coherent and contextually rich, we use the Natural Language Toolkit (NLTK) for accurate sentence tokenization. Specifically, the nltk.tokenize.sent_tokenize function is utilized for its proficiency in detecting natural language boundaries, enabling us to efficiently divide our text into well-defined, paragraph-chunked sentences.

[ ]:
```
# importing the necesary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re
from sklearn.preprocessing import StandardScaler
import docx
import csv
import os
import nltk
import json


nltk.download("punkt")
```

```python
def get_text_chunks(doc_path, min_length=50, max_length=1000):
    # Load the Word document from the specified path
    doc = docx.Document(doc_path)
    # Combine all paragraphs in the document into one string
    full_text = " ".join(para.text for para in doc.paragraphs)

    # Use NLTK to split the full text into sentences
    sentences = nltk.tokenize.sent_tokenize(full_text)

    chunks = []  # Initialize a list to store text chunks
    current_chunk = ""  # Initialize a string to build up a current chunk

    # Iterate through each sentence in the document
    for sentence in sentences:
        # Check if adding this sentence would exceed the max length of a chunk
        if len(current_chunk) + len(sentence) <= max_length:
            current_chunk += (
                " " + sentence
            )  # If not, add the sentence to the current chunk
        else:
            # If the chunk reaches the max length, add it to the list if it meets the␣
↪min length
            if len(current_chunk) >= min_length:
                chunks.append(current_chunk.strip())
            current_chunk = sentence  # Start a new chunk with the current sentence

    # Add the last chunk to the list if it meets the minimum length requirement
    if len(current_chunk) >= min_length:
        chunks.append(current_chunk.strip())

    return chunks  # Return the list of text chunks
```

```
def write_to_csv(chunks, csv_path, topic):
    # Check if the CSV file exists and is non-empty
    file_exists = os.path.isfile(csv_path) and os.path.getsize(csv_path) > 0

    # Open the CSV file in append mode
    with open(csv_path, mode="a", newline="", encoding="utf-8") as file:
        writer = csv.writer(file)  # Create a CSV writer object

        # Write the header row if the file is new or empty
        if not file_exists:
            writer.writerow(["Text", "Source", "Length", "Topic"])

        # Write each chunk as a row in the CSV file
        for chunk in chunks:
            writer.writerow(
                [chunk, "human", len(chunk), topic]
            )  # Include metadata with each chunk


def process_messages_from_json(json_path):
    # Load the JSON file from the specified path
    with open(json_path, "r", encoding="utf-8") as file:
        data = json.load(file)

    messages = []  # Initialize a list to store messages

    # Extract messages from the JSON data
    for message in data["messages"]:
        # Check if the message is of type 'message' and contains text
        if message["type"] == "message" and "text" in message:
            text = message["text"]
```

```
            # If the text is a list, concatenate it into a single string
            if isinstance(text, list):
                text = "".join(
                    [item.get("text", "") for item in text if isinstance(item, dict)]
                    + [item for item in text if isinstance(item, str)]
                )
            messages.append(text)  # Add the processed text to the messages list

    return messages  # Return the list of messages



# Input the topic from the user
topic = input("Please enter the topic: ")

# Specify the path to your Word document and the output CSV file
doc_path = "documents/human reddit_test.docx"
csv_path = "human_reddit.csv"

# Extract text chunks from the Word document and write them to the CSV file
chunks = get_text_chunks(doc_path, min_length=200, max_length=500)
write_to_csv(chunks, csv_path, topic)

print("CSV file has been created with the extracted text chunks.")
```

Section 2: Loading the data

[ ]:
```
 # import statements needed
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[ ]:
```
human_df = pd.read_csv('/content/drive/My Drive/Machine Learning/output.csv')
human_df.head()
human_df.describe()
```

[ ]:
```
            Length
count  2326.000000
mean    314.208512
std     563.752203
min       1.000000
25%      16.000000
50%      36.000000
75%     243.500000
max    3737.000000
```

[ ]:
```
gpt_df = pd.read_csv('/content/drive/My Drive/Machine Learning/gpt_output.csv')
gpt_df.head()
gpt_df.describe()
```

[ ]:
```
            Length
count  2048.000000
mean   1556.297363
std     354.339503
min     584.000000
25%    1389.000000
50%    1442.000000
75%    1485.000000
max    2501.000000
```

Importing external chat responses data.

[ ]:
```
import pandas as pd
import json


def jsonl_to_csv_bytes_from_path(uploaded_jsonl_file_path):
    """Converts uploaded JSONL file at a given path to in-memory CSV bytes.
```

```
    Args:
        uploaded_jsonl_file_path: The file path to the uploaded JSONL file.

    Returns:
        io.BytesIO: CSV data in bytes format.
    """
    # Open the JSONL file and read data into a list of dictionaries
    with open(uploaded_jsonl_file_path, 'r', encoding='utf-8') as file:
        data = [json.loads(line) for line in file]

    # Convert the list of dictionaries to a DataFrame
    df = pd.DataFrame(data)

    # Create a buffer to hold CSV data in bytes format
    from io import BytesIO
    csv_buffer = BytesIO()
    df.to_csv(csv_buffer, index=False, encoding='utf-8')

    # Rewind to the beginning of the buffer and return it
    csv_buffer.seek(0)
    return csv_buffer

# Specify your file path
file_path = '/content/drive/My Drive/Machine Learning/all.jsonl'

# Convert the JSONL file to CSV bytes
csv_bytes = jsonl_to_csv_bytes_from_path(file_path)

# Now you can work with `csv_bytes` as needed, such as reading it into a DataFrame
external_df = pd.read_csv(csv_bytes)
```

[ ]:
```
import pandas as pd


# Creating the original dataframe



# Extracting the 'chatgpt_answers' column, renaming it to 'text', and calculating char␣
 ↪length
external_gpt = external_df[['chatgpt_answers']].copy()
external_gpt.columns = ['Text']
external_gpt['Source'] = 'GPT'
external_gpt['Length'] = external_gpt['Text'].apply(len)
external_gpt['Topic'] = external_df['source'].copy()
external_gpt['Text'] = external_gpt['Text'].str.strip("[]")
external_gpt['Text'] = external_gpt['Text'].str.replace('"', '').str.replace("'", "").
 ↪str.strip()



# Adding a 'source' column and setting all rows to 'GPT'
external_gpt.head()
```

[ ]:

	Text	Source	Length	Topic
0	There are many different best seller lists tha…	GPT	1123	reddit_eli5
1	Salt is used on roads to help melt ice and sno…	GPT	1154	reddit_eli5
2	There are a few reasons why we still have SD (…	GPT	797	reddit_eli5
3	It is generally not acceptable or ethical to a…	GPT	937	reddit_eli5
4	After the Wright Brothers made the first power…	GPT	1242	reddit_eli5

[ ]:
```
external_gpt.describe()
```

[ ]:
```
              Length
count  24322.000000
mean    1126.211290
std      541.041889
min        2.000000
25%      826.000000
50%     1067.000000
75%     1316.000000
max     6305.000000
```

Section 3: Cleaning the text

In our data cleaning process, we employ a series of customized functions aimed at refining the dataset for more efficient analysis and model training. This process is crucial for improving data quality and sharpening the focus of our model.

Initially, we transform the **Source(label)** column into a binary format, where 0 denotes texts authored by humans and 1 indicates texts generated by ChatGPT. This transformation is vital for making the dataset compatible with machine learning algorithms that require numerical input, thereby streamlining data preparation and enhancing the model's efficiency in binary classification tasks.

Furthermore, we normalize all text to lowercase to ensure uniformity across the dataset. This standardization is essential for the algorithm to recognize variations of the same word, such as "Hello," "HELLO," and "hello," as identical, thus simplifying the text data and reducing the feature space. In line with this goal of data standardization, we also eliminate any unecessary spaces, special characters, and null values. These steps are designed to further condense the feature space, increase the accuracy of tokenization, and improve the model's processing efficiency.

[7]:
```
# cleaning the data
import re


def convert_source_labels(df, source_col):
    df[source_col] = df[source_col].apply(
```

```
        lambda x: 0 if x.strip().lower() == "human" else 1 # Convert source labels to␣
 ↪binary
    )
    return df



def lowercase_text(df, text_col):
    df[text_col] = df[text_col].apply(lambda x: x.lower()) # Convert text to lowercase
    return df



def remove_special_characters(df, text_col):
    df[text_col] = df[text_col].apply(lambda x: re.sub(r"[^a-z0-9\s,.?!]", "", x)) #␣
 ↪Remove special characters
    return df



def clean_text(text):
    text = re.sub(r"http\S+", "", text)  # Remove URLs
    text = re.sub(r"\s+", " ", text)  # Replace multiple spaces with a single space
    text = text.strip()  # Remove leading and trailing spaces
    return text



def apply_text_cleaning(df, text_col):
    df[text_col] = df[text_col].apply(clean_text)
    return df

def drop_nan_values(df):
    return df.dropna() # Drop rows with NaN values
```

[ ]:
```
# calling the function to clean the data
human_df = convert_source_labels(human_df, "Source")
human_df = lowercase_text(human_df, "Text")
human_df = remove_special_characters(human_df, "Text")
human_df["Text"] = human_df["Text"].apply(
    clean_text
)
human_df = drop_nan_values(human_df)
```

[ ]:
```
human_df.head()
```

[ ]:

	Text	Source	Length	Topic
0	writing journal entry 6 after this class, writ…	0	1892	journal
1	however, what makes the solo so captivating is…	0	1925	journal
2	furthermore, the guitar being a limited editio…	0	1962	journal
3	observing kawaras art pieces, the concept of h…	0	1422	journal
4	journal entry 8 choose one hc from each of the…	0	1986	journal

Section 4 : Text Generation and Augmentation

Text Generation

To uphold the originality of my text, I embarked on a quest to explore text augmentation strategies tailored to my writing style. As a preliminary step, I aimed to develop an LSTM (Long Short-Term Memory) model designed to emulate my unique manner of expression.

What is a LSTM (Long Short-Term Memory) ?

An LSTM is a recurrent neural network that is designed to avoid the vanishing gradient problem. The vanishing gradient problem to be specific is a challenge encountered in the training of artificial neural networks, particularly evident in traditional recurrent neural networks (RNNs) and deep feedforward networks. It occurs when the gradients of the network's loss function decrease exponentially as they are propagated backward through the network's layers during training. This gradient diminution leads to very small updates to the weights of the network's early layers, which effectively halts the learning process for these parts of the model.

How does it work

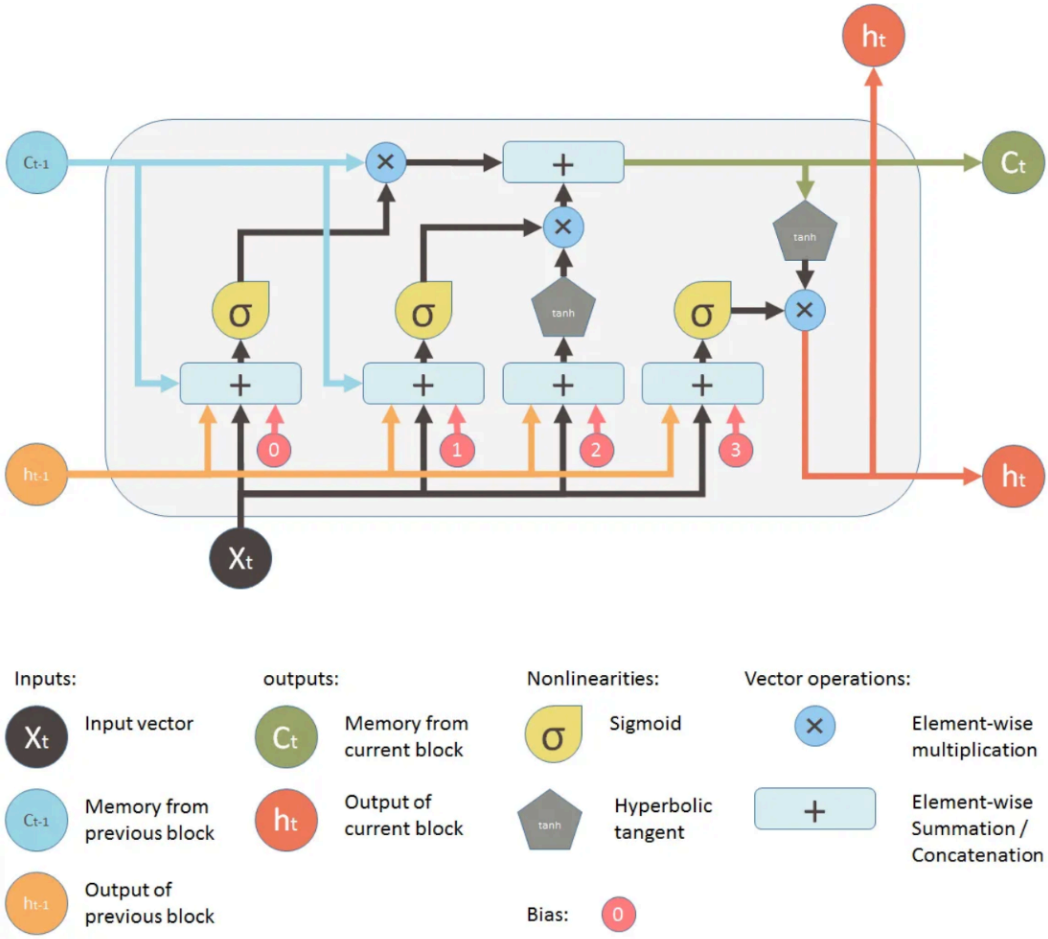


Figure 4

Source. Architectural representation of a LSTM unit.

From the image above the blue line represents the memory from the previous block and this is what we denote as the 'long term' memory in our LSTM model. Normally the blue line will not have any weights and biases to ensure that our gradient does not explode or vanish. Conversely the orange line represents our short-term memory and the black circle represents our input vector.

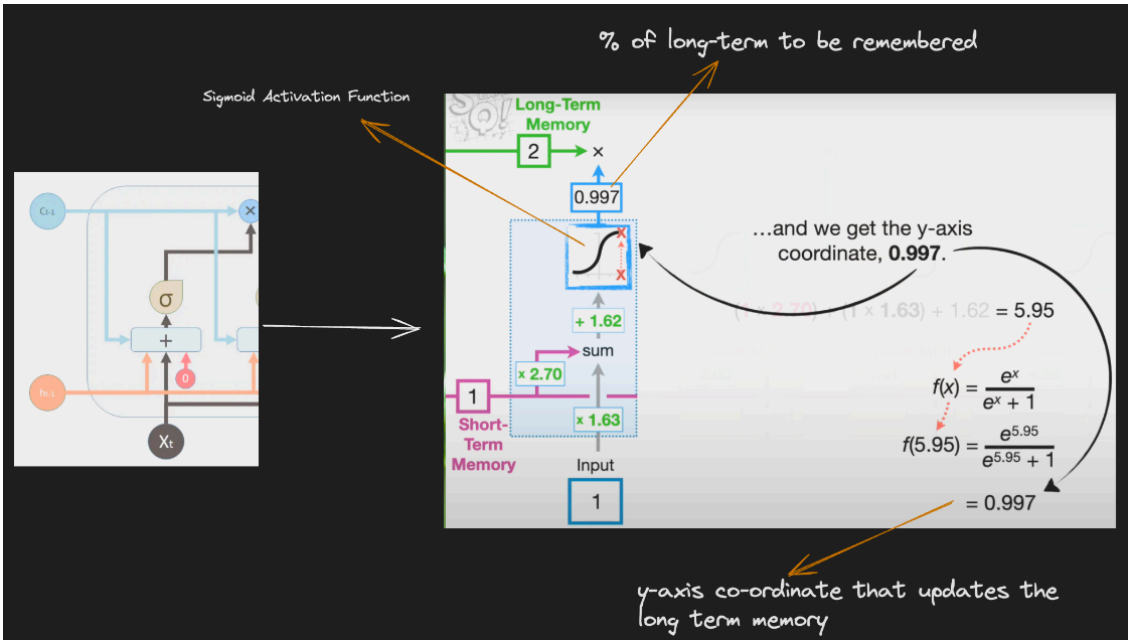


Figure 5

Low-level representation of the first segment of an LSTM unit.

Getting more granular in LSTMs, what essentially happens is that the input gate's activation involves combining the current input (x_t) and the previous short-term memory (h_{t-1}), each multiplied by their respective weights plus a bias term, and then applying a sigmoid activation function. This can be mathematically represented as follows:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

Note that the sigmoid activation function maps our x-axis coordinate to a y-axis coordinate between 0 and 1.

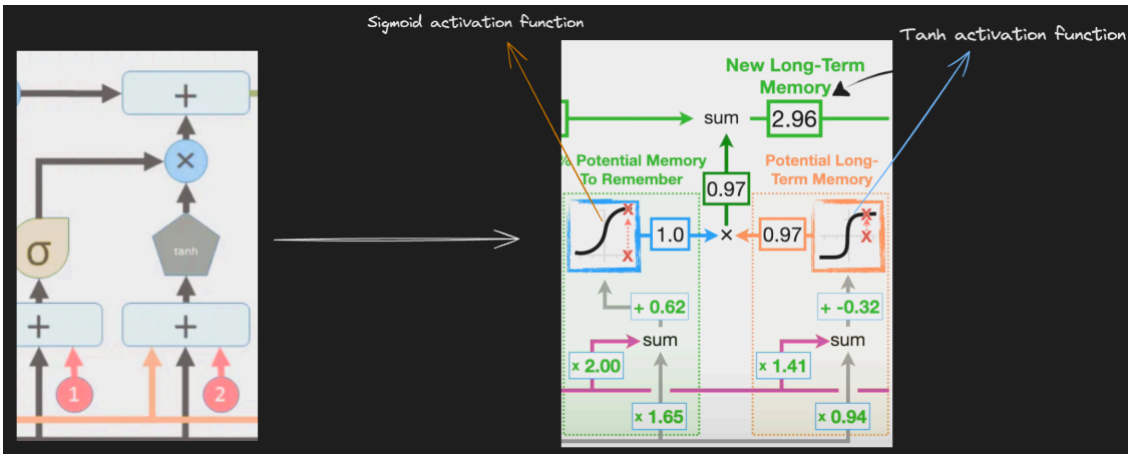


Figure 6

Low-level representation of the second segment of an LSTM unit.

In this second LSTM segment, we still take the short-term memory (h_{t-1}) and current input (x_t) multiplied by their respective weights. However, in this case, we are interested in calculating the new long-term memory, (C_t).

This is done by two block sections (the ones in green and orange in the image above), where one calculates the potential long-term memory, whereas the other contributes to what percentage of the long-term memory to remember.

This calculation is orchestrated by two critical block sections (green and orange in Figure 6 above), serving distinct yet complementary roles:

1. **The candidate long-term memory creation(Orange block)** which employs the ($tanh$) activation function to generate a vector of candidate values, ($tildeC_t$), for updating the long-term memory. This step is crucial for introducing non-linearity and ensuring that the potential updates to the long-term memory are normalized within a range of -1 to 1.

2. **The memory update gate** which determines the extent to which each unit of the long-term memory is updated. This involves calculating what percentage of the old long-term memory (C_{t-1}) to retain versus how much of the new candidate memory ($tildeC_t$) to add.

The integration of these components leads to the formulation of the new long-term memory as follows:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

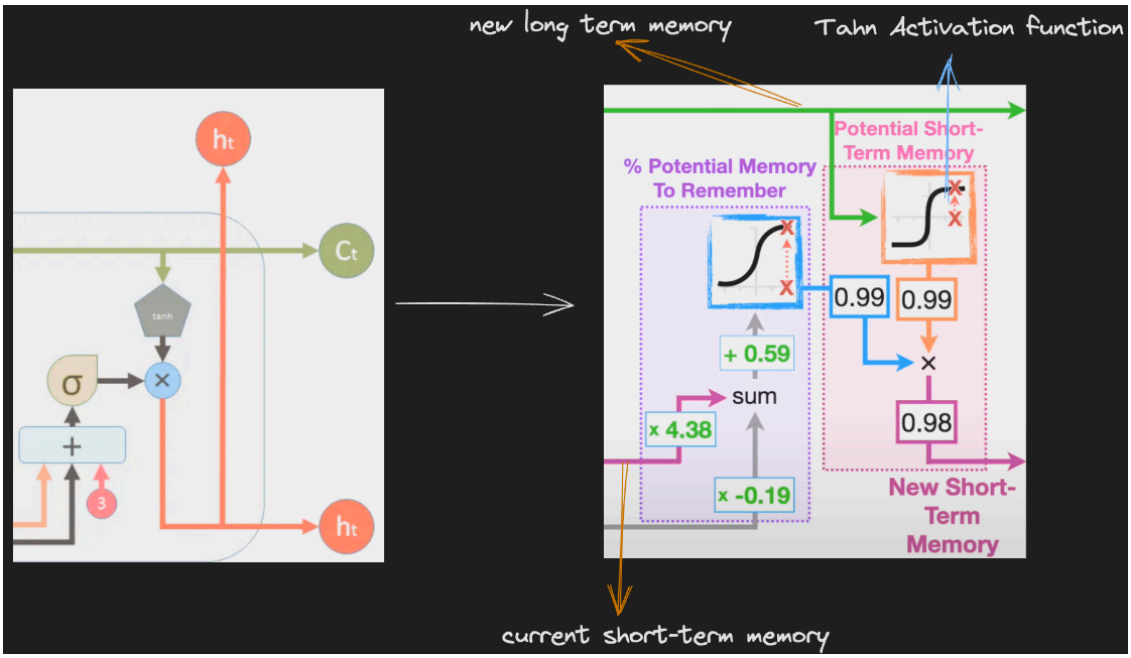


Figure 7

Low-level representation of the third segment of an LSTM unit.

Finally, in this third segment our focus is to derive a new short-term memory. We do this by taking the new-long term memory and use it as an input in the Tahn activation function. Moreover to determine the % of the potential short-term memory to retain we use the same approach we used in segment two to determine the % of the long-term memory to retain.

In our case of generating text that mimics my writing style, LSTM units synergize to update and utilize memories, effectively capturing and emulating the unique nuances of the desired style. By expanding the sequence with additional LSTM units, the model learns from longer contexts, ensuring the output remains consistent and coherent.

Results of the LSTM model

As one can observe in our case, our LSTM model performs very poorly, only managing a 0.5601 accuracy. This is evident from the sample output of me attempting to generate some text data.

> **Seed:** "exploring the depths of machine learning, one model at a time, unraveling the mysteries within data."
>
> **Generated output:** the context of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the contract of the cont

Generated output with some randomization: looks what hesrlenry filyensing rogt gorco and linao landuawe the clanse between srowed in the reaieiment moic work uotitya dilent types for even and nomes, oy risk parsinn and toullen. serure ouport Done.

[ ]:
```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, B
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.callbacks import EarlyStopping
```

[ ]:
```
# Tokenization
human_text = ' '.join(human_df['Text'].astype(str))
tokenizer = Tokenizer(char_level=True, filters='')
tokenizer.fit_on_texts(human_text)
sequences = tokenizer.texts_to_sequences([human_text])[0]


# Create overlapping sequences
seq_length = 100  # Initial sequence length for experimentation
step = 1
sentences = []
next_chars = []
for i in range(0, len(sequences) - seq_length, step):
    sentences.append(sequences[i: i + seq_length])
    next_chars.append(sequences[i + seq_length])
n_vocab = len(tokenizer.word_index) + 1


# Prepare the dataset
X = np.reshape(sentences, (len(sentences), seq_length, 1)) / float(n_vocab)
y = to_categorical(next_chars, num_classes=n_vocab)
```

```
# Define the LSTM model
model = Sequential([
    LSTM(128, input_shape=(X.shape[1], X.shape[2]), return_sequences=True),
    Dropout(0.2),
    LSTM(128),
    Dropout(0.2),
    Dense(n_vocab, activation='softmax'),
])

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='loss', patience=5)

# Model training
model.fit(X, y, epochs=15, batch_size=64, callbacks=[early_stopping])

# Save the model and tokenizer
model.save('lstm_model.h5')
tokenizer_json = tokenizer.to_json()
with open('tokenizer.json', 'w', encoding='utf-8') as f:
    f.write(tokenizer_json)
```

```
Epoch 1/15
38485/38485 [==============================] - 473s 12ms/step - loss: 2.1332 -
accuracy: 0.3743
Epoch 2/15
38485/38485 [==============================] - 465s 12ms/step - loss: 1.7709 -
accuracy: 0.4795
Epoch 3/15
38485/38485 [==============================] - 458s 12ms/step - loss: 1.6798 -
accuracy: 0.5047
Epoch 4/15
```

```
38485/38485 [==============================] - 457s 12ms/step - loss: 1.6312 -
accuracy: 0.5184
Epoch 5/15
38485/38485 [==============================] - 460s 12ms/step - loss: 1.5979 -
accuracy: 0.5277
Epoch 6/15
38485/38485 [==============================] - 463s 12ms/step - loss: 1.5749 -
accuracy: 0.5342
Epoch 7/15
38485/38485 [==============================] - 471s 12ms/step - loss: 1.5565 -
accuracy: 0.5390
Epoch 8/15
38485/38485 [==============================] - 455s 12ms/step - loss: 1.5414 -
accuracy: 0.5435
Epoch 9/15
38485/38485 [==============================] - 454s 12ms/step - loss: 1.5287 -
accuracy: 0.5467
Epoch 10/15
38485/38485 [==============================] - 451s 12ms/step - loss: 1.5171 -
accuracy: 0.5492
Epoch 11/15
38485/38485 [==============================] - 456s 12ms/step - loss: 1.5079 -
accuracy: 0.5525
Epoch 12/15
38485/38485 [==============================] - 446s 12ms/step - loss: 1.4996 -
accuracy: 0.5547
Epoch 13/15
38485/38485 [==============================] - 460s 12ms/step - loss: 1.4923 -
accuracy: 0.5566
Epoch 14/15
38485/38485 [==============================] - 461s 12ms/step - loss: 1.4851 -
accuracy: 0.5582
Epoch 15/15
```

```
38485/38485 [==============================] - 469s 12ms/step - loss: 1.4795 -
accuracy: 0.5601
```

[ ]:
```python
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.text import tokenizer_from_json
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

# Load model
model = load_model('lstm_model.h5')

# Load tokenizer
with open('tokenizer.json', 'r', encoding='utf-8') as f:
    tokenizer_json = f.read()
```

[ ]:
```python
import numpy as np
import random
import sys
import json
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Parse the JSON string into a Python dictionary
tokenizer_data = json.loads(tokenizer_json)

# Access the 'word_index' inside the 'config' dictionary
char_to_index = tokenizer_data["config"]["word_index"]
char_to_index = eval(char_to_index)
print(char_to_index)

# Creating the inverse mapping from indices to characters
index_to_char = {}
for char, index in char_to_index.items():
```

```
    index_to_char[index] = char

# Load your trained model
model = load_model('lstm_model.h5')

# Helper function to prepare the seed text
def prepare_seed(pattern):
    """Converts seed text to a list of integers."""
    pattern = [char_to_index[char] for char in pattern]
    return pattern

def generate_text(model, char_to_index, index_to_char, seed_text, num_generate=1000):
    """Generates new text from a trained LSTM model."""
    pattern = prepare_seed(seed_text)

    print(f'Seed: "{seed_text}"')

    # Generate characters
    for _ in range(num_generate):
        x = np.reshape(pattern, (1, len(pattern), 1))
        x = x / float(len(char_to_index))  # Normalize

        prediction = model.predict(x, verbose=0)[0]
        index = np.argmax(prediction)

        result = index_to_char[index]
        sys.stdout.write(result)

        pattern.append(index)
        pattern = pattern[1:len(pattern)]
    print("\nDone.")
```

```
# Example usage
seed_text = "exploring the depths of machine learning, one model at a time, unraveling␣
 ↪the mysteries within data."
  # Provide your own seed text
num_generate = 200  # Number of characters to generate

# Ensure your seed text at least matches the expected sequence length for the model
generate_text(model, char_to_index, index_to_char, seed_text, num_generate)
```

```
{' ': 1, 'e': 2, 't': 3, 'i': 4, 'a': 5, 'n': 6, 'o': 7, 's': 8, 'r': 9, 'c':
10, 'l': 11, 'h': 12, 'd': 13, 'u': 14, 'm': 15, 'p': 16, 'g': 17, 'f': 18, 'y':
19, 'v': 20, 'b': 21, 'w': 22, ',': 23, '.': 24, 'k': 25, 'x': 26, 'q': 27, 'z':
28, 'j': 29, '0': 30, '2': 31, '1': 32, '3': 33, '5': 34, '4': 35, '9': 36, '6':
37, '8': 38, '7': 39, '?': 40, '!': 41}
Seed: "exploring the depths of machine learning, one model at a time, unraveling
the mysteries within data."
 the context of the contract of the contract of the contract of the contract of
the contract of the contract of the contract of the contract of the contract of
the contract of the contract of the contract of the cont
Done.
```

[ ]:
```
def sample(preds, temperature=1.0):
    # Convert predictions to probabilities
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds + 1e-10) / temperature  # Avoid log(0) by adding a small␣
 ↪constant
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)

# randmozied text generation
def generate_text(model, char_to_index, index_to_char, seed_text, num_generate=1000):
```

```
    """Generates new text from a trained LSTM model."""
    pattern = prepare_seed(seed_text)

    print(f'Seed: "{seed_text}"')

    # Generate characters
    for _ in range(num_generate):
        x = np.reshape(pattern, (1, len(pattern), 1))
        x = x / float(len(char_to_index))  # Normalize

        prediction = model.predict(x, verbose=0)[0]
        index = sample(prediction)

        result = index_to_char[index]
        sys.stdout.write(result)

        pattern.append(index)
        pattern = pattern[1:len(pattern)]
    print("\nDone.")

# Example usage
seed_text = "exploring the depths of machine learning, one model at a time, unraveling␣
 ↪the mysteries within data."
  # Provide your own seed text
num_generate = 200  # Number of characters to generate

# Ensure your seed text at least matches the expected sequence length for the model
generate_text(model, char_to_index, index_to_char, seed_text, num_generate)
```

```
Seed: "exploring the depths of machine learning, one model at a time, unraveling
the mysteries within data."
 books what hesrlenry filyensing rogt gorco and linao landuawe the clanse
between srowed in the reaieiment moic work uotitya dilent types for even and
```

```
nomes, oy risk parsinn and toullen. serure ouport
Done.
```

Generative model 2: distilgpt2(lightweight GPT2)

In an attempt to improve we use a distilled version of GPT which we fine-tune with our current dataset of human text and the results were slightly better though the text was still not coherent.

> **Seed:** "computer science"
>
> **Generated output:** I will have a very different opinion about the climate change debate and more on its future for you if it turns out true here at Google or Amazon (or Twitter). We were given various questions concerning our past years of research into global warming because in part due to my own personal experience since many people from one country are aware that we did not "win". And with today's data changing around us without any consequences all over the world - those who don't think so as far as getting themselves included include myself!

With these attempts not bearing any fruits and it being extremely computationally expensive to generate thousands of new text samples I settle for utilizing an already optimized model for my augmentation.

[ ]:
```
# using pre-trained models
from transformers import GPT2LMHeadModel, GPT2Tokenizer


model_name = "gpt2"  # You can choose other models like "gpt2-medium", "gpt2-large"␣
 ↪depending on your needs and resources.


tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)
```

[ ]:
```
from transformers import GPT2Tokenizer


tokenizer = GPT2Tokenizer.from_pretrained('distilgpt2')


# Assign the EOS token as padding token
tokenizer.pad_token = tokenizer.eos_token


# Assuming 'Text' column contains the text you want to fine-tune on
```

```
texts = human_df['Text'].tolist()
encodings = tokenizer(texts, max_length=512, truncation=True, padding="max_length",␣
↪return_tensors="pt")
```

[ ]:
```
from torch.utils.data import Dataset, DataLoader


class TextDataset(Dataset):
    def __init__(self, encodings):
        self.encodings = encodings


    def __getitem__(self, idx):
        return {key: val[idx] for key, val in self.encodings.items()}


    def __len__(self):
        return len(self.encodings.input_ids)


dataset = TextDataset(encodings)
```

[ ]:
```
from transformers import GPT2LMHeadModel, AdamW, get_linear_schedule_with_warmup
import torch
import os
from google.colab import drive  # For Google Colab


# Mount your Google Drive for saving
drive.mount('/content/gdrive')


# -------------------- Model Loading and Preparation --------------------
model = GPT2LMHeadModel.from_pretrained('distilgpt2')
model.train()


# -------------------- Dataset and Dataloaders --------------------
# ... You'll need to replace these with how you create your datasets and loaders
train_loader = DataLoader(dataset, batch_size= 8, shuffle=True)
```

```
# ------------------- Optimization Setup -------------------
optimizer = AdamW(model.parameters(), lr=5e-5)
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=500,␣
 ↪num_training_steps=-1)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# ------------------- Training and Saving -------------------
epochs = 4
save_directory = '/content/drive/My Drive/finetuned_model'
save_interval = 2  # Save every 2 epochs

for epoch in range(epochs):
    for batch in train_loader:
        optimizer.zero_grad()
        inputs, labels = batch['input_ids'].to(device), batch['input_ids'].to(device)
        outputs = model(inputs, labels=labels)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
        scheduler.step()

    print(f"Epoch: {epoch}, Train Loss: {loss.item()}")

    # Save the model at specified intervals
    if (epoch + 1) % save_interval == 0:
        if not os.path.exists(save_directory):
            os.makedirs(save_directory)
        model.save_pretrained(save_directory)
```

[ ]:
```python
from transformers import GPT2Tokenizer, GPT2Model
prompt = "computer science"
input_ids = tokenizer.encode(prompt, return_tensors='pt').to(device)

# Create the attention mask (assuming your device handles PyTorch tensors)
attention_mask = torch.ones(input_ids.shape).to(device)

# Generate text
output_sequences = model.generate(
    input_ids=input_ids,
    attention_mask=attention_mask,  # Include the attention mask
    max_length=500,
    temperature=1.0,
    top_k=50,
    top_p=0.95,
    repetition_penalty=1.2,
    do_sample=True,
    num_return_sequences=1
)

generated_text = tokenizer.decode(output_sequences[0], skip_special_tokens=True)
print(generated_text)
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

computer science. In this post, I>d add something special to consider:
-I will have a very different opinion about the climate change debate and more
on its future for you if it turns out true here at Google or Amazon (or
Twitter). We were given various questions concerning our past years of research
into global warming because in part due to my own personal experience since many
people from one country are aware that we did not "win". And with today's data
changing around us without any consequences all over the world - those who don't
think so as far as getting themselves included include myself!
```

Text Augmentation

To augment the text, I employ three distinct techniques designed to generate a diverse array of augmented text versions, each contributing uniquely to enhancing the dataset for model training.

- The first technique involves the use of **Contextual Word Embeddings Augmentation**, leveraging distilbert-base-uncased for substituting words in the text with their contextual embeddings. This approach modifies approximately 65% of the text, introducing synonyms and contextually relevant replacements that maintain the original meaning while varying the expression.
- The second technique, **TextGenie Augmentation (textgenie)**, utilizes a paraphrasing model, specifically ramsrigouthamg/t5_paraphraser combined with bert-base-uncased, to reformulate sentences. This method is weighted more heavily in my augmentation pipeline since it provided very good augmented samples in comparison to the other two models.
- Lastly, **Back Translation Augmentation (bt)** employs a two-step translation process using facebook/wmt19-en-de and facebook/wmt19-de-en models for English to German and German back to English translations, respectively. This technique translates the text to a different language and then back to the original language. While the core information is preserved, subtle nuances in translation introduce variability in phrasing and sentence structure, contributing further to the diversity of the augmented text.

Using this pipeline I was able to generate **3098** new augmented samples significantly hacking up my human text class.

[ ]:
```
# Set the locale
!pip install textgenie
```

[ ]:
```
# Texdt augmentation - synonyms
!pip install nlpaug
import nltk
nltk.download('wordnet')
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

[ ]: True

[ ]:
```
!pip install nlpaug torch transformers sacremoses pandas textgenie
```

[ ]:
```python
import pandas as pd
import nlpaug.augmenter.word as naw
import random
from textgenie import TextGenie
from transformers import AutoTokenizer
import time
from tqdm import tqdm  # Import tqdm

# Assuming df is your DataFrame and it has been loaded

# Set up the models and tokenizer
aug_emb = naw.ContextualWordEmbsAug(
    model_path='distilbert-base-uncased',
    action='substitute',
    aug_p=0.65
)

textgenie = TextGenie("ramsrigouthamg/t5_paraphraser", 'bert-base-uncased')

model_en_de = "facebook/wmt19-en-de"
model_de_en = "facebook/wmt19-de-en"
aug_bt = naw.BackTranslationAug(
    from_model_name=model_en_de,
    to_model_name=model_de_en
)

tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")

augmented_rows = []
total_rows = len(human_df)
csv_file_path = 'augmented_text_versions.csv'
```

```
# Ensure the file is empty or set up with the correct headers initially
pd.DataFrame(columns=['Text']).to_csv(csv_file_path, index=False)

def truncate_text(text, max_length=512):
    tokens = tokenizer.tokenize(text)
    if len(tokens) > max_length - 2:
        tokens = tokens[:max_length - 2]
    return tokenizer.convert_tokens_to_string(tokens)

# Wrap human_df.iterrows() with tqdm for a progress bar
for index, row in tqdm(human_df.iterrows(), total=total_rows, desc="Processing rows"):
    text = row['Text']
    truncated_text = truncate_text(text)

    choice = random.choices(
        population=["emb", "textgenie", "bt"],
        weights=[0.25, 0.5, 0.25],
        k=1
    )[0]

    if choice == "emb":
        augmented_text = aug_emb.augment(truncated_text)
        augmented_rows.append({'Text': augmented_text})
    elif choice == "textgenie":
        augmented_texts = textgenie.augment_sent_t5(truncated_text, "paraphrase: ",␣
↪n_predictions=2)
        for aug_text in augmented_texts[:2]:
            augmented_rows.append({'Text': aug_text})
    elif choice == "bt":
        augmented_text = aug_bt.augment(truncated_text)
        augmented_rows.append({'Text': augmented_text})
```

```
    # Every 100 rows, append to the CSV and clear the list
    if (index + 1) % 100 == 0 or (index + 1) == total_rows:
        pd.DataFrame(augmented_rows).to_csv(csv_file_path, mode='a', header=False,␣
↪index=False)
        augmented_rows.clear()  # Clear the list after writing
```

[ ]:
```
human_augmented_df = pd.read_csv('/content/drive/My Drive/Machine Learning/
↪augmented_text_versions.csv')
human_augmented_df.describe()
```

[ ]:
```
        Text
count   3098
unique  2959
top     ['']
freq      79
```

[ ]:
```
# Create the 'Source' column
human_augmented_df['Source'] = 'human'


# Create the 'Length' column
human_augmented_df['Length'] = human_augmented_df['Text'].apply(lambda x: len(str(x)))


# Create the 'Topic' column
human_augmented_df['Topic'] = 'augmented'


# Clean the 'Text' column to ensure all are strings
human_augmented_df['Text'] = human_augmented_df['Text'].astype(str)


# Remove rows with NaN values in 'Text' column
human_augmented_df.dropna(subset=['Text'], inplace=True)
human_augmented_df.head()
```

[ ]:

	Text	Source	Length	Topic
0	Upon studying the elements of music, write an …	human	335	augmented
1	Writing journal entry 6 after this class, writ…	human	241	augmented
2	what makes the solo so captivating is the voca…	human	353	augmented
3	What makes the solo so captivating is the voca…	human	401	augmented
4	['Furthermore, the limited edition guitar crea…	human	963	augmented

[ ]:
```
# putting all our data together
master_data = pd.concat([human_df,human_augmented_df,gpt_df,␣
 ↪external_gpt],ignore_index=True)
master_data.describe()
```

[ ]:

	Length
count	31794.000000
mean	998.038529
std	632.199816
min	1.000000
25%	657.000000
50%	1019.000000
75%	1328.000000
max	6305.000000

[ ]:
```
#calling the function to clean the data
master_data = convert_source_labels(master_data, "Source")
master_data = lowercase_text(master_data, "Text")
master_data = remove_special_characters(master_data, "Text")
master_data["Text"] = master_data["Text"].apply(
    clean_text
)
master_data = drop_nan_values(master_data)
```

[ ]:
```
master_data.head()
```

[ ]:

	Text	Source	Length	Topic
0	writing journal entry 6 after this class, writ…	0	1892	journal
1	however, what makes the solo so captivating is…	0	1925	journal
2	furthermore, the guitar being a limited editio…	0	1962	journal
3	observing kawaras art pieces, the concept of h…	0	1422	journal
4	journal entry 8 choose one hc from each of the…	0	1986	journal

[ ]:
```
master_data.describe()
```

[ ]:

	Source	Length
count	31296.000000	31296.000000
mean	0.826847	991.583429
std	0.378386	634.994635
min	0.000000	1.000000
25%	1.000000	649.750000
50%	1.000000	1010.000000
75%	1.000000	1313.000000
max	1.000000	6305.000000

Section 5 : Task Discussion

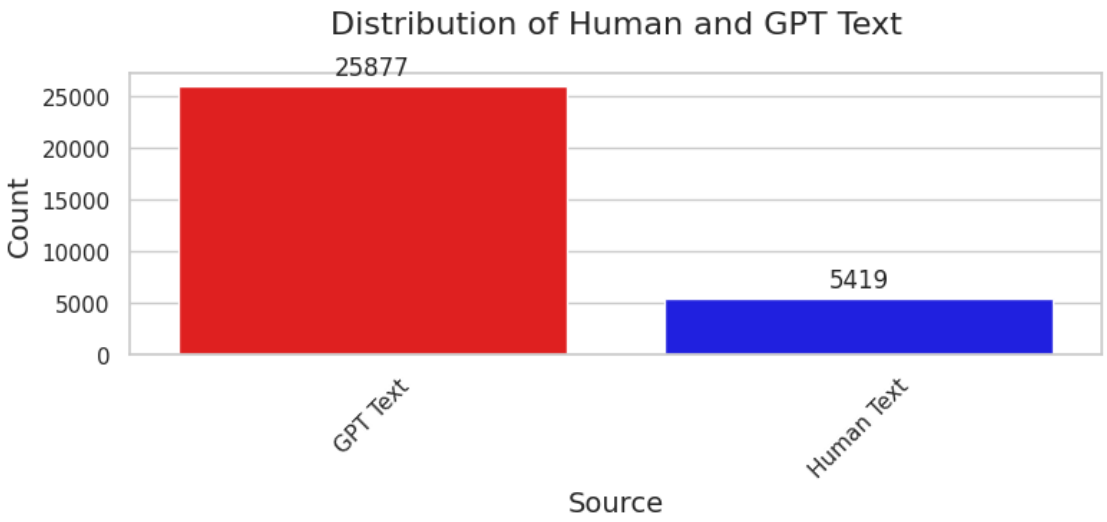
Specifically, I frame the problem as a supervised classification task, where the objective is to learn a mapping between a representation of the text and a binary variable. This variable is 0 if the text is generated by a human (me in this case), and 1 if the text is generated by ChatGPT. More formally, by utilizing machine learning strategies, I seek to learn a function (f) that, given an input text represented as a set of features ($[f_1 i, ..., f_k i]$), outputs an estimated label ($l_i \in \{1, 0\}$), $i.e.$, ($l_i = f(t_i)$).

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd


def plot_class_distribution(df, source_column, title, colors=['red', 'blue'], mapping
    = None):
    """
```

```
    Creates a bar plot showing the distribution of values in a given column of a␣
↪DataFrame.

    Args:
        df (pd.DataFrame): The DataFrame containing the data.
        source_column (str): The name of the column containing the classes.
        title (str): The title for the bar plot.
        colors (list, optional): A list of colors to use for the bars. Defaults to␣
↪['red', 'blue'].
    """

    df_temp = df.copy()

    if mapping is not None:
        df_temp[source_column] = df_temp[source_column].map(mapping)

    class_counts = df_temp[source_column].value_counts().reset_index()
    class_counts.columns = ['source', 'count']

    sns.set(style="whitegrid")
    plt.figure(figsize=(8, 4))

    barplot = sns.barplot(x='source', y='count', data=class_counts, palette=colors)

    plt.xlabel('Source', fontsize=14)
    plt.ylabel('Count', fontsize=14)
    plt.title(title, fontsize=16, y=1.1)

    for p in barplot.patches:
        height = p.get_height()
        barplot.annotate(f'{int(height)}',
                         xy=(p.get_x() + p.get_width() / 2, height),
```

```
                        xytext=(0, 3),
                        textcoords="offset points",
                        ha='center', va='bottom')

    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()



plot_class_distribution(master_data, 'Source', 'Distribution of Human and GPT Text',␣
  ↪mapping = {0: 'Human Text', 1: 'GPT Text'})
```

```
<ipython-input-44-11e180126a2c>:27: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  barplot = sns.barplot(x='source', y='count', data=class_counts,
palette=colors)
```



Balancing the dataset (undersampling)

Before we proceed with further data preprocessing, it's evident from the statistical analysis of our datasets that there's a disproportionate amount of data from ChatGPT compared to human-generated data. This forces us to use a **random undersampling strategy** with the intention of balancing our data since our model would be biased towards predicting the majority class (ChatGPT text), affecting its ability to accurately identify human-generated text.

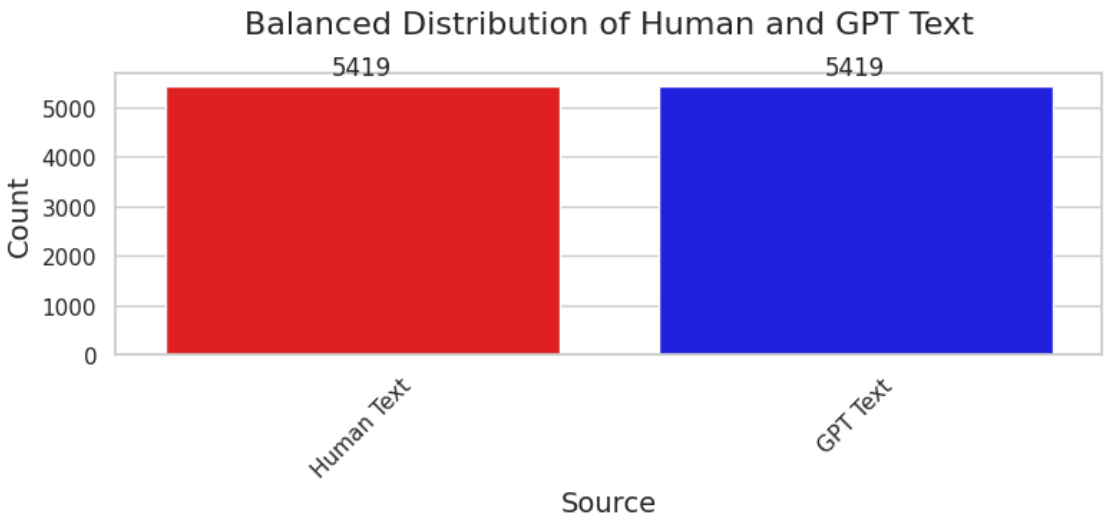
[3]:
```python
from imblearn.under_sampling import RandomUnderSampler
import pandas as pd


def balance_data(df, text_column, target_column, random_state=42):
    """
    Balances a DataFrame using random under-sampling.

    Args:
        df (pd.DataFrame): The DataFrame to balance.
        text_column (str): The name of the column containing the text data.
        target_column (str): The name of the column containing the target labels.
        random_state (int, optional): Random seed for reproducibility. Defaults to 42.

    Returns:
        pd.DataFrame: A new DataFrame with balanced classes.
    """

    X = df[[text_column]]
    y = df[target_column]

    rus = RandomUnderSampler(random_state=random_state)
    X_res, y_res = rus.fit_resample(X, y)

    return pd.DataFrame(X_res, columns=[text_column]).assign(**{target_column: y_res})
```

[ ]:
```
master_balanced = balance_data(master_data, "Text", "Source")
master_balanced.describe()
```

[ ]:
```
             Source
count  10838.000000
mean       0.500000
std        0.500023
min        0.000000
25%        0.000000
50%        0.500000
75%        1.000000
max        1.000000
```

[ ]:
```
plot_class_distribution(master_balanced, 'Source', 'Balanced Distribution of Human and␣
↪GPT Text', mapping = {0: 'Human Text', 1: 'GPT Text'})
```

```
<ipython-input-44-11e180126a2c>:27: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  barplot = sns.barplot(x='source', y='count', data=class_counts,
palette=colors)
```



Tokenization and Vectorization

With our dataset now balanced, the next step is to tokenize and vectorize the data. This process converts our data into a format that our model can readily ingest and understand

TF-IDF. This method provides a quantitative measure to evaluate the significance of words within a document's context, compared against a broader corpus.

Reason we are using this

In the context of differentiating between ChatGPT and human text, the TF-IDF framework aids in uncovering nuanced linguistic features that could signal the origin of a text. For instance, certain complex syntactic structures or specialized vocabulary might be more prevalent in human-generated content, whereas ChatGPT might exhibit distinct repetitive patterns or adherence to common phrases as shown in the image below.

By converting textual data into a matrix of TF-IDF features, we equip our classification models with the ability to learn and predict based on these quantifiable differences. The implementation of TF-IDF in this classification problem not only enhances model accuracy but also provides insights into the linguistic and stylistic nuances distinguishing AI-generated text from human prose.

How it works

At the heart of TF-IDF lies the Term Frequency (TF), a measure quantifying the occurrence frequency of a term within a given document. The underlying premise suggests a direct correlation between a term's frequency in a document and its importance to that document. Mathematically, TF is represented as:

$$TF(term) = (\frac{\text{Number of times the term appears in the document}}{\text{Total number of terms in the document}})$$

This normalization factor accounts for document length variability, ensuring that longer documents do not unfairly influence the importance of terms due to their sheer size.

While TF focuses on a term's local significance, the Inverse Document Frequency (IDF) metric evaluates a term's commonality across the entire document corpus. IDF mitigates the influence of frequently occurring terms, thus amplifying the significance of rarer terms across documents. The IDF is calculated using the formula:

$$IDF(term) = \log\left(\frac{\text{Total number of documents in the corpus}}{\text{Number of documents containing the term}}\right)$$

This logarithmic scale prevents terms that are common across the corpus from diminishing the value of more distinctive terms. Finally, the TF-IDF of a term is calculated by multiplying TF and IDF scores.

$$TF - IDF(term) = TF(term) \times IDF(term)$$

[24]:
```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
import joblib


def create_tfidf_features(df, text_column, labels=None):
  """
  Creates TF-IDF features from a DataFrame.

  Args:
      df (pd.DataFrame): The DataFrame containing the text data.
      text_column (str): The name of the column containing the text data.
      labels (pd.Series, optional): The target labels (if applicable). Defaults to␣
↪None.

  Returns:
      tuple: A tuple containing the TF-IDF vectorizer, the TF-IDF matrix, and the␣
↪target labels (if provided).
  """
```

```
  vectorizer = TfidfVectorizer()
  tfidf_matrix = vectorizer.fit_transform(df[text_column])
  joblib.dump(vectorizer, 'tfidf_vectorizer.pkl')

  if labels is not None:
      return vectorizer, tfidf_matrix, labels.values
  else:
      return vectorizer, tfidf_matrix
```

[ ]:
```
vectorizer, tfidf_matrix, labels = create_tfidf_features(master_balanced, 'Text',␣
 ↪master_balanced['Source'])

# Access the TF-IDF matrix and labels (if provided)
print(tfidf_matrix.toarray())
print(labels)
```

```
[[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 …
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]]
[0 0 0 … 1 1 1]
```

Section 6: Model Selection

Our model selection this time was significantly guided by existing research on this subject. A noteworthy study published in 2023, which tackled the same exploratory question, conducted an experiment with 11 different models. From their results, Extremely Randomized Trees seemed to perform the best with an accuracy score of 0.77. However, it is important to note that all other models were within a 0.03 standard deviation which is not a significant performance difference(Islam et al., n.d.). From this analysis we select the Extremely Randomized Trees and Gradient Boosting as our models of choice.

Extremely Randomized Trees

What are extremely randomized trees ?

Extra trees, short for extremely randomized trees, is a supervised machine learning technique that constructs an ensemble of unpruned decision or regression trees following the conventional top-down approach. Similar to random forests, it generates multiple decision trees, but with a twist: it randomly samples the data for each tree without replacement and employs the entire learning sample to develop the trees, instead of relying on bootstrap replicas. This method enhances the diversity among the trees, contributing to the overall robustness and accuracy of the model.

How does it work ?

The core idea revolves around creating multiple decision trees (an ensemble), where each tree is trained on the full dataset. However, the way each tree is grown introduces variability. For each tree, a random set of features is selected at each node, and then, rather than calculating the optimal split point for these features, split points are also chosen randomly.

Regarding the splitting of the data, let S be a subset of the dataset at a node. For a given feature a, the algorithm selects a split point $\hat{a}_c$ at random. The selection of $\hat{a}_c$ is uniform within the range $[a\text{min}, a\text{max}]$, where a_{min} and a_{max} are the minimum and maximum values of feature a in subset S, respectively.

The dataset S at the node is then split into two subsets based on this randomly chosen $\hat{a}_c$:

$$S_{\text{left}} = x \in S \mid x_a < \hat{a}_c$$

$$S_{\text{right}} = x \in S \mid x_a \geq \hat{a}_c$$

With the splitting approach explained, all the generated individual trees t_i in an Extra-Trees ensemble may not always make the most effective splits, introducing variability in performance. Nonetheless, the ensemble $T = t_1, t_2, \ldots, t_M$ harnesses these diverse trees to improve classification accuracy through collective decision-making. Thus the ensemble's prediction $\hat{y}$ for an input x emerges from majority voting:

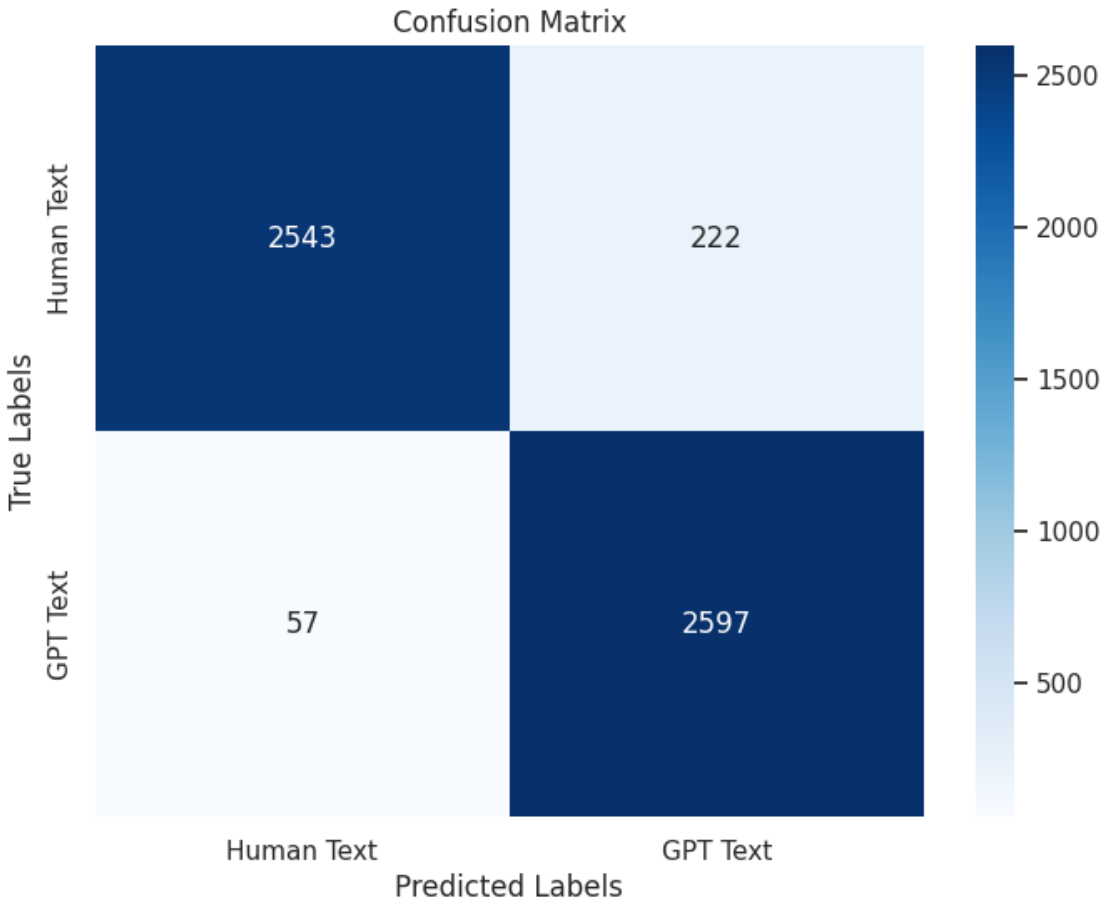
$$\hat{y} = \text{mode}t_1(x), t_2(x), \ldots, t_M(x)$$

Finally,it is important to note that extra trees do not optimize a loss function directly and thus depend on the law of large numbers through ensemble trees to achieve robust generalization and accuracy.

[ ]:
```
# Extremely random trees
from sklearn.model_selection import train_test_split
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import joblib
from google.colab import files

# Assuming 'tfidf_matrix' is your matrix of TF-IDF features and 'labels' are your target labels

# Splitting the data into training and testing sets with a 50/50 split
X_train, X_test, y_train, y_test = train_test_split(tfidf_matrix, labels, test_size=0.5, random_state=42)

# Creating an instance of the ExtraTreesClassifier
extratrees = ExtraTreesClassifier(n_estimators=100, random_state=42, criterion='entropy')

# Training the model on the training data
extratrees.fit(X_train, y_train)

# Making predictions on the testing data
predictions = extratrees.predict(X_test)

# Evaluating the model's performance
accuracy = accuracy_score(y_test, predictions)
print(f'Accuracy: {accuracy}')
print(classification_report(y_test, predictions))



# Save the model to disk
joblib.dump(extratrees, 'extratrees_model.pkl')
```

```
print("Model saved as 'extratrees_model.pkl'.")


files.download('extratrees_model.pkl')
print("Model downloaded to local device")


# Generating the confusion matrix
conf_matrix = confusion_matrix(y_test, predictions)


# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Human Text',␣
 ↪'GPT Text'], yticklabels=['Human Text', 'GPT Text'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
Accuracy: 0.9485144860675402
              precision    recall  f1-score   support

           0       0.98      0.92      0.95      2765
           1       0.92      0.98      0.95      2654

    accuracy                           0.95      5419
   macro avg       0.95      0.95      0.95      5419
weighted avg       0.95      0.95      0.95      5419

Model saved as 'extratrees_model.pkl'.
<IPython.core.display.Javascript object>
<IPython.core.display.Javascript object>
Model downloaded to local device
```



Given we utilize the model from sklearn library here is some pseudocode on how the model works under the hood.

Input: Training set S containing features extracted from text and labels indicating ChatGPT or human authorship.

Output: Tree ensemble $T = \{t_1, t_2, ..., t_M\}$ for classification.

Algorithm Steps:

1. **Initialize** an empty ensemble T.
2. **For** $i = 1$ **to** M **do**:
   - t_i = BuildAnExtraTree(S).
   - Add t_i to the ensemble T.
3. **Return** T.

Function: BuildAnExtraTree(S). Input: Training set S.

Output: A decision tree t.

- **If** $|S| < n_{\min}$, or all candidate attributes are constant, or the class labels (ChatGPT vs. human) are constant in S:
  - **Return** a leaf node labeled by class frequencies (the probability distribution of ChatGPT vs. human in S).
- **Else**:
  - Use the SplitANode(S) to obtain the best split s^*.
  - **If** s^* is not none:
    - Split S into two subsets S_{left} and S_{right} according to s^*.
    - $t_{\text{left}} = \text{BuildAnExtraTree}(S_{\text{left}})$.
    - $t_{\text{right}} = \text{BuildAnExtraTree}(S_{\text{right}})$.
    - **Return** a node with split s^* having t_{left} and t_{right} as children.
  - **Else**:
    - **Return** a leaf node labeled by class frequencies in S.

Function: SplitANode(S). Input: The local learning subset S corresponding to the node we want to split.

Output: A split $[a < a_c]$ or nothing.

- **If** StopSplit(S) is TRUE, then **return** nothing.
- Otherwise, select K attributes $\{a_1, ..., a_K\}$ among all non-constant (in S) candidate attributes.
- Draw K splits $\{s_1, ..., s_K\}$, where $s_i = \text{PickARandomSplit}(S, a_i)$, $\forall i = 1, ..., K$.
- **Return** a split s^* such that $\text{Score}(s^*, S) = \max_{i=1,...,K} \text{Score}(s_i, S)$.

Function: PickARandomSplit(S, a). Input: Subset S, Attribute a.

Output: A split s.

- **If** a is numerical:
  - Find $a_{\min}$ and $a_{\max}$, the minimal and maximal values of a in S.
  - Draw a cut-point a_c uniformly in $[a_{\min}, a_{\max}]$.
  - **Return** the split $[a < a_c]$.
- **If** a is categorical:
  - Determine A_S, the subset of possible values of a that appear in S.
  - Randomly choose a subset A_1 of A_S and a subset A_2 of the complement of A_S.
  - **Return** the split $[a \in A_1 \cup A_2]$.

Function: StopSplit(S). Input: A subset S.

Output: A boolean.

- **If** $|S| < n_{\min}$, then **return** TRUE.
- **If** all attributes are constant in S, then **return** TRUE.
- **If** the output is constant in S, then **return** TRUE.
- **Otherwise**, **return** FALSE.

[ ]:
```
import joblib


# Load the model from the file
extratrees_model = joblib.load('extratrees_model.pkl')
vectorizer = joblib.load('tfidf_vectorizer.pkl')


from sklearn.feature_extraction.text import TfidfVectorizer


# New text to classify
new_texts = ["Ensure each section is substantiated with relevant legal texts, case␣
 ↪law, academic commentary, and statistical evidence where available. Use␣
 ↪authoritative sources from the EU, national competition authorities, and legal␣
 ↪scholars to support your arguments. This will not only enrich your analysis but also␣
 ↪demonstrate the depth of your research to your professor."]


# Transform the new text using the loaded vectorizer
tfidf_new_texts = vectorizer.transform(new_texts)


# Make predictions with the loaded model
predictions = extratrees_model.predict(tfidf_new_texts)


# Print the predictions
print(predictions)
```

```
[0]
```

Model 2: Gradient Boosting(XGBoost)

This is a supervised learning algorithm that attempts to accurately predict a target by combining the estimates of simpler or weaker learners by learning sequentially and converting many weak learners into a complex learner. You can think of these learners as trees as we did in the Extra trees example above. Additionally the algorithm is called gradient boosting because it utilizes a gradient descent procedure to minimize the loss when adding new learners to the ensemble.

How does it work?

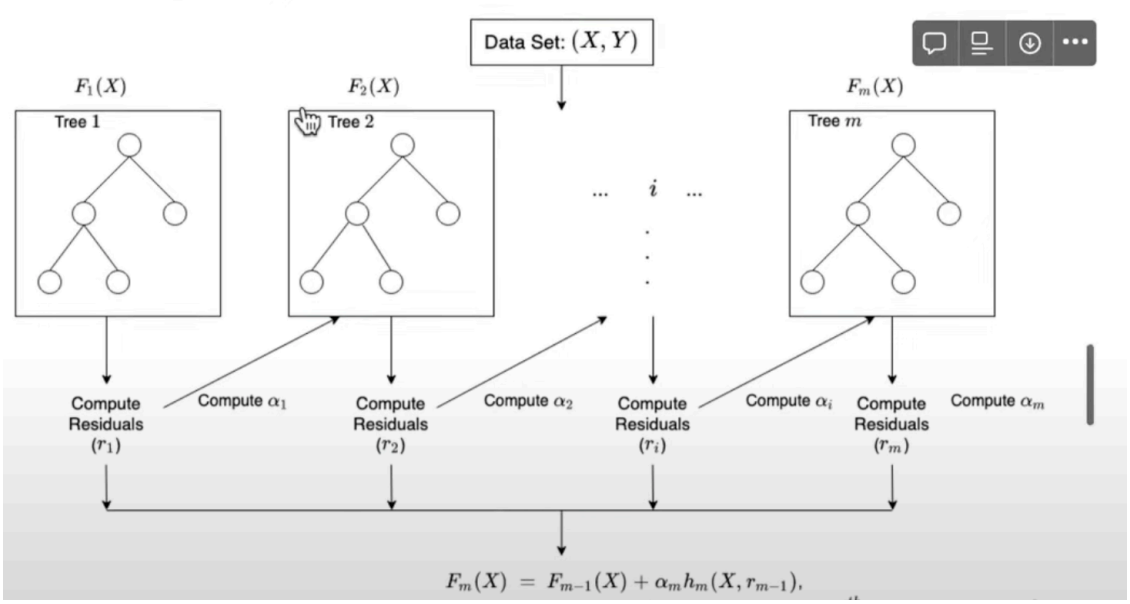


Figure 8

Visual representation of how Gradient boosting works with tree ensembles

We first start with a simple model to return a constant value. We shall call this $(F_0(X))$. For each subsequent tree $(i = 1, \ldots, m)$ where (m) is the predefined maximum number of trees, we first compute the residual (r_i) for the particular tree given by:

$$[r_i = -\frac{\partial L(Y, F(x))}{\partial F(x)} \quad \text{where} \quad F(x) = F_{i-1}(x)]$$

Where $(L(Y, F(x)))$ is a differentiable equation for a loss function. For our case, we choose to use the mean squared error (MSE) which is given by $((Y - F(X))^2)$. Where:

- $(F(X))$ is the prediction of the previous tree $(F_{i-1}(x))$.

Using this residual, we then attempt to fit a new subsequent tree with the idea of correcting the errors of the previous model (reducing the residuals). This is given by $[h_i(X, r_{i-1}) \quad \nabla r_i]$ where: - (∇r_i) is the gradient of (r_i) with respect to the prediction $(F(X))$.

- $(h_i(X, r_{i-1}))$ is a function to predict (r_i) (how much adjustment needs to be made to improve the previous tree).

Having the residuals, we can now update the prediction of our current tree (i). This is given by

$$[F_i(x) = F_{i-1}(x) + \alpha \cdot h_i(X, r_{i-1})]$$

where: - $(F_{i-1}(x))$ is the prediction from the previous tree - (λ) is the learning rate - $(h_i(X, r_{i-1}))$ is the (i_{th}) base learner that is added to the model during the (i_{th}) iteration, trained on the previous trees' residuals as calculated above.

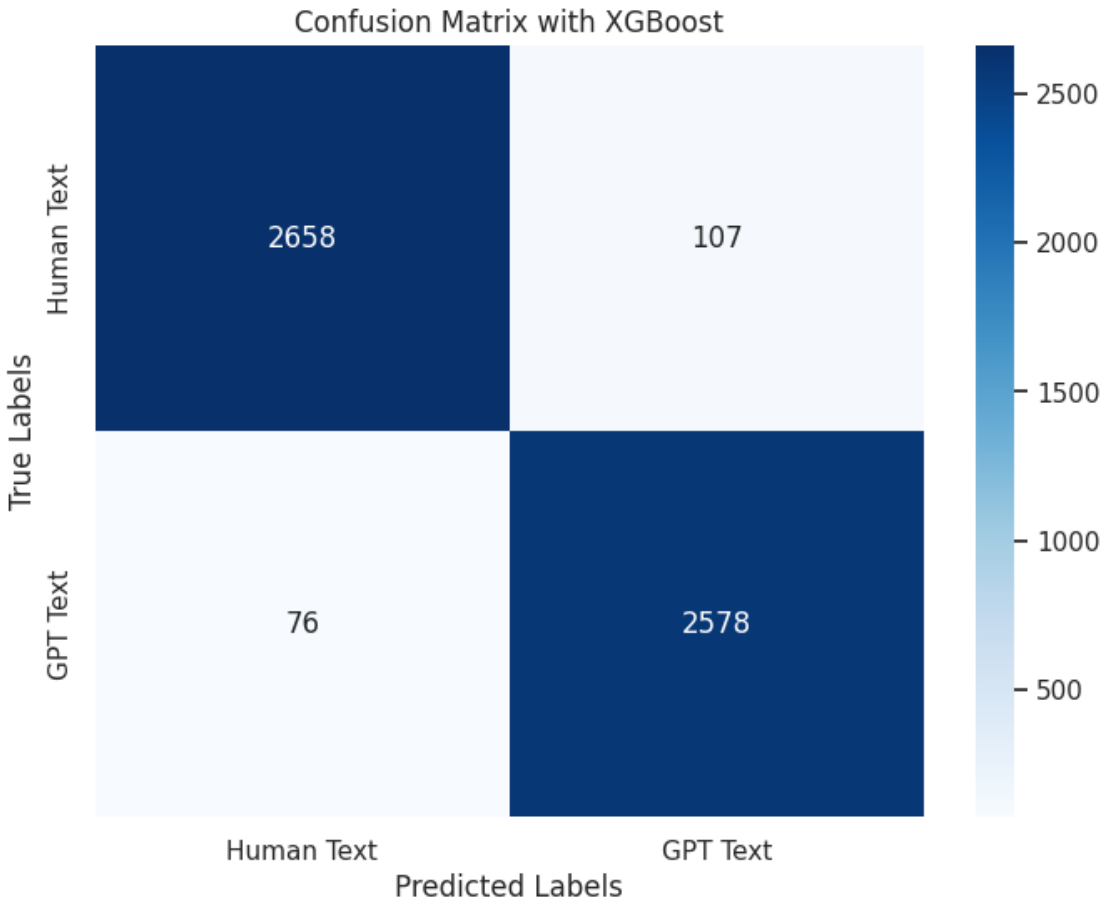
Finally, the above steps happen for all the trees generated until we reach the final tree (m) where the final output $F_m(X)$ is given by the weighted sum of the collection of trees.

[ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import joblib
from google.colab import files
import xgboost as xgb
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming 'tfidf_matrix' is your matrix of TF-IDF features and 'labels' are your␣
 ↪target labels

# Splitting the data into training and testing sets with a 50/50 split
X_train, X_test, y_train, y_test = train_test_split(tfidf_matrix, labels, test_size=0.
 ↪5, random_state=42)

# Creating an instance of the XGBClassifier
xgb_model = xgb.XGBClassifier(n_estimators=50, random_state=42,␣
 ↪use_label_encoder=False, eval_metric='logloss', verbosity=1, reg_alpha= 0.1,␣
 ↪reg_lambda=2)

# Training the model on the training data
xgb_model.fit(X_train, y_train)
```

```
# Making predictions on the testing data
predictions = xgb_model.predict(X_test)

# Evaluating the model's performance
accuracy = accuracy_score(y_test, predictions)
print(f'Accuracy: {accuracy}')
print(classification_report(y_test, predictions))

# Save the model to disk
joblib.dump(xgb_model, 'xgb_model.pkl')
print("Model saved as 'xgb_model.pkl'.")

# Downloading the model file
files.download('xgb_model.pkl')
print("Model downloaded to local device")

# Generating the confusion matrix
conf_matrix = confusion_matrix(y_test, predictions)

# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Human Text',␣
↪'GPT Text'], yticklabels=['Human Text', 'GPT Text'])
plt.title('Confusion Matrix with XGBoost')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
Accuracy: 0.9662299317217199
              precision    recall  f1-score   support

           0       0.97      0.96      0.97      2765
```

```
           1       0.96      0.97      0.97      2654

    accuracy                           0.97      5419
   macro avg       0.97      0.97      0.97      5419
weighted avg       0.97      0.97      0.97      5419

Model saved as 'xgb_model.pkl'.

<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>

Model downloaded to local device
```

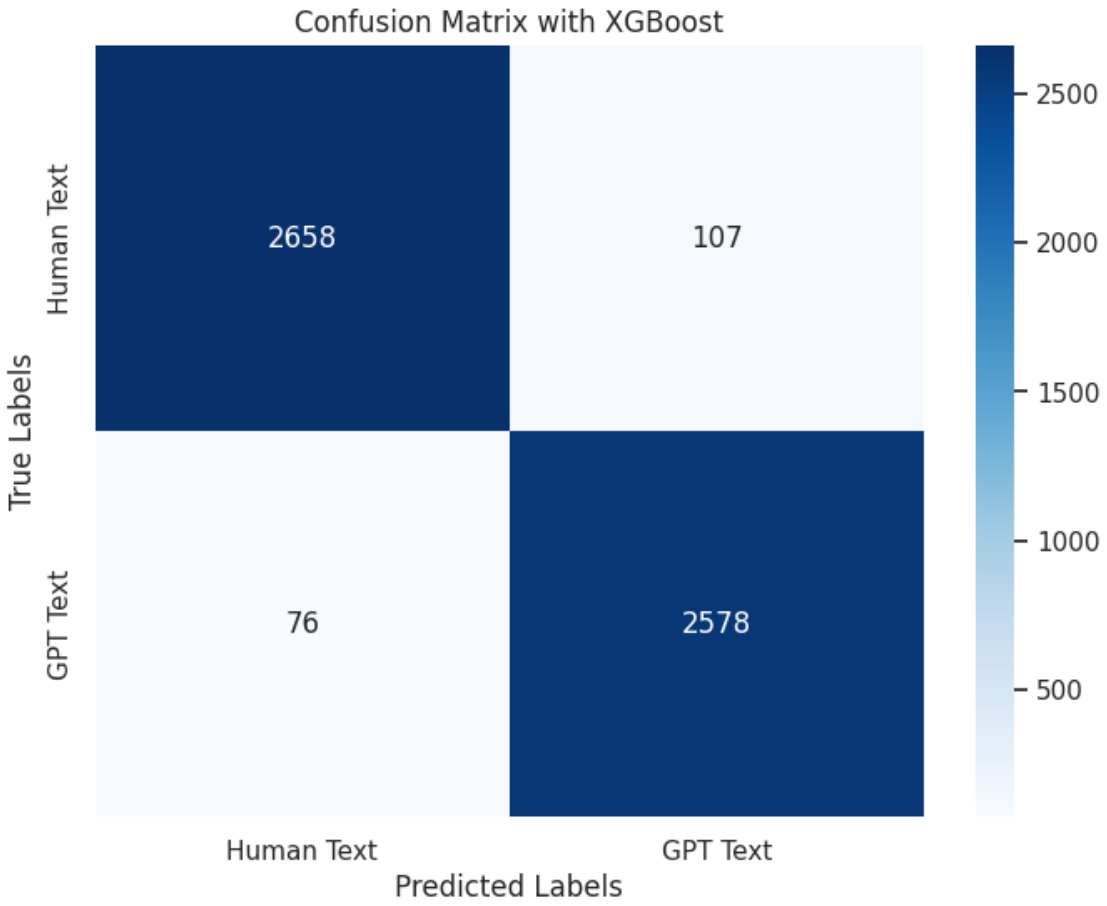


[ ]:
```
from sklearn.model_selection import train_test_split, KFold, cross_val_score,
 ↪GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import joblib
from google.colab import files
import xgboost as xgb
import matplotlib.pyplot as plt
import seaborn as sns


# Assuming 'tfidf_matrix' is your matrix of TF-IDF features and 'labels' are your
 ↪target labels


# Splitting the data into training and testing sets with a 50/50 split
cv = KFold(n_splits=5, shuffle=True, random_state=42)

# Creating an instance of the XGBClassifier
xgb_model = xgb.XGBClassifier(n_estimators=50, random_state=42,
 ↪use_label_encoder=False, eval_metric='logloss', verbosity=1, reg_alpha= 0.1,
 ↪reg_lambda=2)


# Evaluate using cross-validation
scores = cross_val_score(xgb_model, tfidf_matrix, labels, scoring='accuracy', cv=cv)

# Report cross-validated accuracy
print(f"Cross-validation Accuracy (Mean): {scores.mean():.3f}")
print(f"Cross-validation Accuracy (Standard Deviation): {scores.std():.3f}")



# Training the model on the training data
xgb_model.fit(X_train, y_train)

# Making predictions on the testing data
```

```
predictions = xgb_model.predict(X_test)


# Evaluating the model's performance
accuracy = accuracy_score(y_test, predictions)
print(f'Accuracy: {accuracy}')
print(classification_report(y_test, predictions))


# Save the model to disk
joblib.dump(xgb_model, 'xgb_model.pkl')
print("Model saved as 'xgb_model.pkl'.")


# Downloading the model file
files.download('xgb_model.pkl')
print("Model downloaded to local device")


# Generating the confusion matrix
conf_matrix = confusion_matrix(y_test, predictions)


# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Human Text',␣
 ↪'GPT Text'], yticklabels=['Human Text', 'GPT Text'])
plt.title('Confusion Matrix with XGBoost')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
Cross-validation Accuracy (Mean): 0.973
Cross-validation Accuracy (Standard Deviation): 0.003
Accuracy: 0.9662299317217199
              precision    recall  f1-score   support

           0       0.97      0.96      0.97      2765
```

```
           1       0.96      0.97      0.97      2654

    accuracy                           0.97      5419
   macro avg       0.97      0.97      0.97      5419
weighted avg       0.97      0.97      0.97      5419

Model saved as 'xgb_model.pkl'.
<IPython.core.display.Javascript object>
<IPython.core.display.Javascript object>
Model downloaded to local device
```



External Use of Human data

Given my concerns about overfitting and the unexpectedly high performance of our results, I opted to incorporate additional external data from human-generated sources. This decision was made to assess if the limited size of my dataset was a contributing factor to the observed performance anomalies. The link to the dataset can be found here:

Kaggle Dataset

[ ]:
```
# adding more data to the human text to see if it performs better so
ex_df = pd.read_csv('/content/Training_Essay_Data.csv')
ex_df.describe()
```

[ ]:

	generated
count	29145.000000
mean	0.399279
std	0.489759
min	0.000000
25%	0.000000
50%	0.000000
75%	1.000000
max	1.000000

[ ]:
```
ex_df.head()
external_human_df = ex_df[ex_df['generated'] == 0]
external_human_df.describe()
external_human_df.head()
```

[ ]:

	text	generated
749	Cars. Cars have been around since they became …	0
750	Transportation is a large necessity in most co…	0
751	"America's love affair with it's vehicles seem…	0
752	How often do you ride in a car? Do you drive a…	0
753	Cars are a wonderful thing. They are perhaps o…	0

[ ]:
```
# renaming columns
external_human_df = external_human_df.rename(columns={'text': 'Text', 'generated':␣
↪'Source'})



# Replace 'Source' column
external_human_df['Source'] = external_human_df['Source'].replace(0, 'human')



# Create the 'Length' column
external_human_df ['Length'] = external_human_df ['Text'].apply(lambda x: len(str(x)))

# Create the 'Topic' column
external_human_df ['Topic'] = 'external_human'

# Clean the 'Text' column to ensure all are strings
external_human_df ['Text'] = external_human_df ['Text'].astype(str)

# Remove rows with NaN values in 'Text' column
external_human_df .dropna(subset=['Text'], inplace=True)

external_human_df.head()
```

[ ]:

	Text	Source	Length	\
749	cars. cars have been around since they became …	human	3277	
750	transportation is a large necessity in most co…	human	2721	
751	americas love affair with its vehicles seems t…	human	4400	
752	how often do you ride in a car? do you drive a…	human	3979	
753	cars are a wonderful thing. they are perhaps o…	human	4665	

	Topic
749	external_human
750	external_human

```
751  external_human
752  external_human
753  external_human
```

[ ]:
```
#calling the function to clean the data
external_human_df = convert_source_labels(external_human_df, "Source")
external_human_df = lowercase_text(external_human_df, "Text")
external_human_df = remove_special_characters(external_human_df, "Text")
external_human_df["Text"] = external_human_df["Text"].apply(
    clean_text
)
external_human_df = drop_nan_values(external_human_df)
```

[ ]:
```
external_human_df.head()
```

[ ]:
```
                                                  Text  Source  Length  \
749  cars. cars have been around since they became …       0    3277
750  transportation is a large necessity in most co…       0    2721
751  americas love affair with its vehicles seems t…       0    4400
752  how often do you ride in a car? do you drive a…       0    3979
753  cars are a wonderful thing. they are perhaps o…       0    4665

             Topic
749  external_human
750  external_human
751  external_human
752  external_human
753  external_human
```

[ ]:
```
external_human_df.describe()
```

[ ]:
```
         Source        Length
count  17508.0  17508.000000
mean       0.0   2381.829278
```

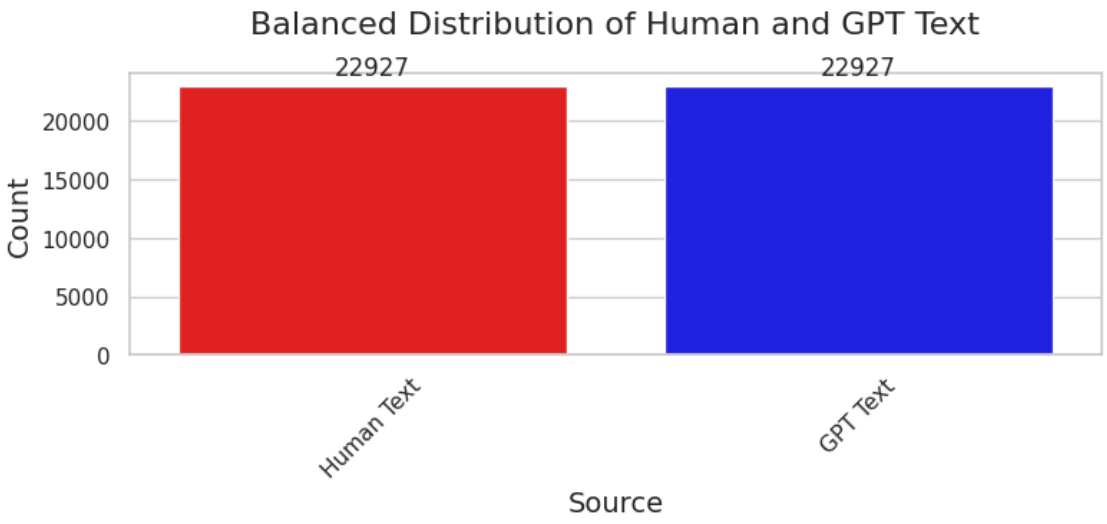
```
std        0.0   1032.153341
min        0.0    237.000000
25%        0.0   1605.000000
50%        0.0   2244.500000
75%        0.0   2946.000000
max        0.0   9119.000000
```

[ ]:
```
master_data_lg = pd.concat([master_data, external_human_df])
master_data_lg.describe()
```

[ ]:
```
             Source        Length
count  66312.000000  66312.000000
mean       0.390231   1725.701668
std        0.487806   1111.083714
min        0.000000      1.000000
25%        0.000000    991.000000
50%        0.000000   1455.000000
75%        1.000000   2379.250000
max        1.000000   9119.000000
```

[ ]:
```
new_master_balanced = balance_data(master_data, 'Text', 'Source')
plot_class_distribution(new_master_balanced, 'Source', 'Balanced Distribution of Human␣
 ↪and GPT Text', mapping = {0: 'Human Text', 1: 'GPT Text'})
```

```
<ipython-input-44-11e180126a2c>:27: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  barplot = sns.barplot(x='source', y='count', data=class_counts,
palette=colors)
```



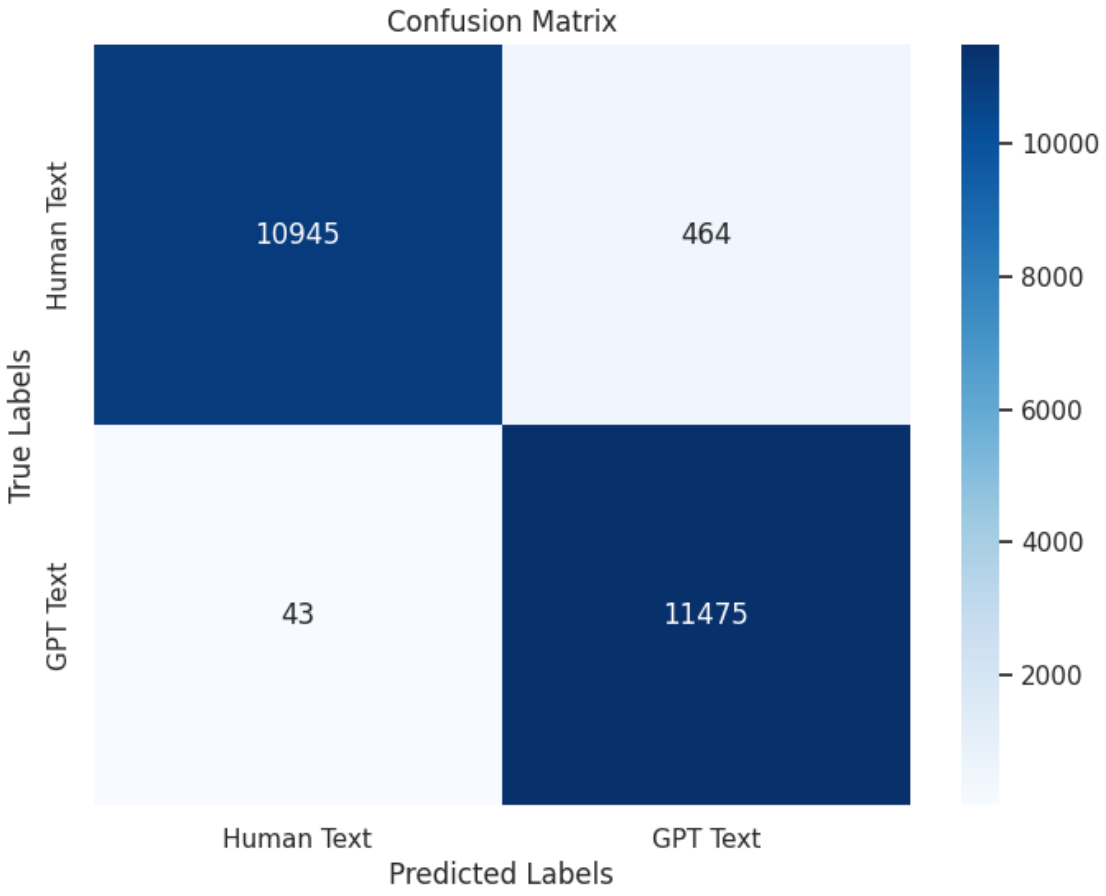
[ ]:
```
vectorizer, tfidf_matrix, labels = create_tfidf_features(new_master_balanced, 'Text',␣
↪new_master_balanced['Source'])
```

[ ]:
```
# Extremely random trees
from sklearn.model_selection import train_test_split
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import joblib
from google.colab import files

# Splitting the data into training and testing sets with a 50/50 split
X_train, X_test, y_train, y_test = train_test_split(tfidf_matrix, labels, test_size=0.
↪5, random_state=42)

# Creating an instance of the ExtraTreesClassifier
extratrees = ExtraTreesClassifier(n_estimators=100, random_state=42,␣
↪criterion='entropy')

# Training the model on the training data
extratrees.fit(X_train, y_train)
```

```
# Making predictions on the testing data
predictions = extratrees.predict(X_test)

# Evaluating the model's performance
accuracy = accuracy_score(y_test, predictions)
print(f'Accuracy: {accuracy}')
print(classification_report(y_test, predictions))



# Save the model to disk
joblib.dump(extratrees, 'extratrees_model.pkl')
print("Model saved as 'extratrees_model.pkl'.")

files.download('extratrees_model.pkl')
print("Model downloaded to local device")

# Generating the confusion matrix
conf_matrix = confusion_matrix(y_test, predictions)

# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Human Text',␣
↪'GPT Text'], yticklabels=['Human Text', 'GPT Text'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
Accuracy: 0.9778863348889955
              precision    recall  f1-score   support

           0       1.00      0.96      0.98     11409
```

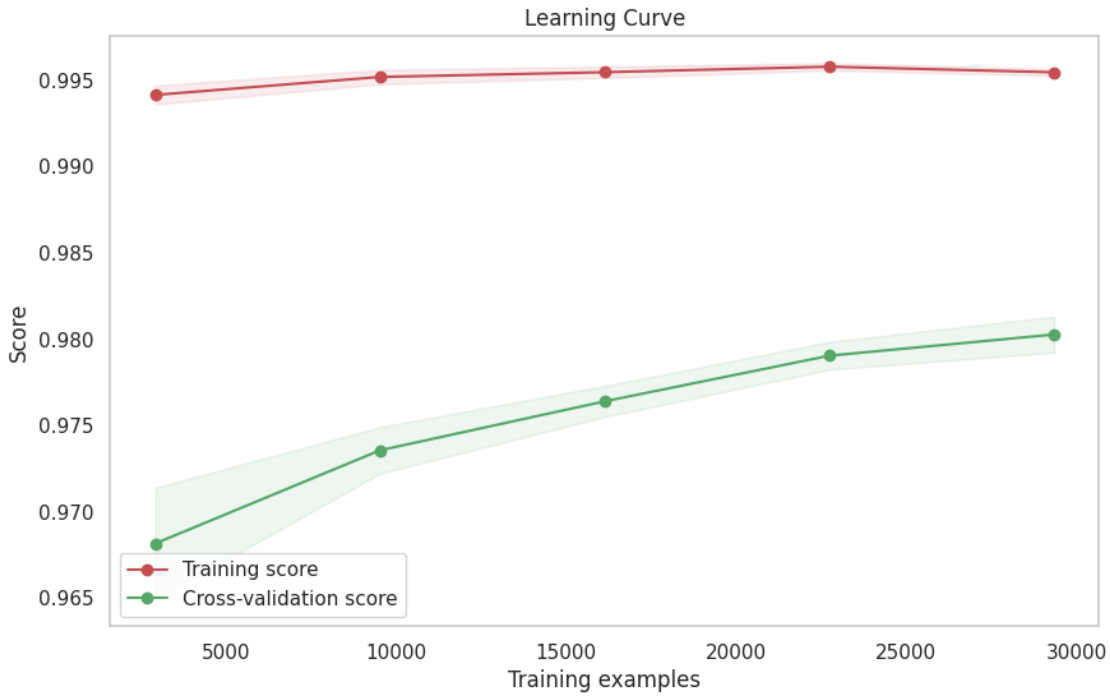
```
           1       0.96      1.00      0.98     11518

    accuracy                           0.98     22927
   macro avg       0.98      0.98      0.98     22927
weighted avg       0.98      0.98      0.98     22927

Model saved as 'extratrees_model.pkl'.

<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>

Model downloaded to local device
```



[ ]:
```
from sklearn.model_selection import cross_val_score, learning_curve, train_test_split
from sklearn.ensemble import ExtraTreesClassifier
import matplotlib.pyplot as plt
import numpy as np

# Assuming 'tfidf_matrix' and 'labels' are defined elsewhere in your code

# Splitting the data (keeping a test set aside to check final model performance)
X_train, X_test, y_train, y_test = train_test_split(tfidf_matrix, labels, test_size=0.
 ↪2, random_state=42)

# Creating an instance of the ExtraTreesClassifier
extratrees = ExtraTreesClassifier(n_estimators=100, random_state=42,␣
 ↪criterion='entropy')

# Cross-validation to evaluate model performance
cv_scores = cross_val_score(extratrees, X_train, y_train, cv=5)
print(f'Cross-Validation Scores: {cv_scores}')
print(f'Mean CV Score: {np.mean(cv_scores)}')

# Training and plotting a learning curve to check for overfitting
train_sizes, train_scores, validation_scores = learning_curve(extratrees, X_train,␣
 ↪y_train, train_sizes=np.linspace(0.1, 1.0, 5), cv=5, scoring='accuracy')

# Calculating mean and standard deviation for train and validation sets
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
validation_scores_mean = np.mean(validation_scores, axis=1)
validation_scores_std = np.std(validation_scores, axis=1)

# Plotting the learning curve
plt.figure(figsize=(10, 6))
```

```
plt.grid()


plt.fill_between(train_sizes, train_scores_mean - train_scores_std, train_scores_mean␣
 ↪+ train_scores_std, color="r", alpha=0.1)
plt.fill_between(train_sizes, validation_scores_mean - validation_scores_std,␣
 ↪validation_scores_mean + validation_scores_std, color="g", alpha=0.1)


plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training score")
plt.plot(train_sizes, validation_scores_mean, 'o-', color="g", label="Cross-validation␣
 ↪score")


plt.title("Learning Curve")
plt.xlabel("Training examples")
plt.ylabel("Score")
plt.legend(loc="best")


plt.show()
```

```
Cross-Validation Scores: [0.97901049 0.97901049 0.98119122 0.9815976
0.98037077]
Mean CV Score: 0.9802361174424188
```



Testing our models with External data

external_testing_df = pd.read_csv('test_essays.csv')

[4]:
```
import pandas as pd


external_testing_df = pd.read_csv("Test_Essay_Data.csv")
external_testing_df.head()
```

[4]:

	text	generated
0	Car-free cities have become a subject of incre…	1
1	Car Free Cities Car-free cities, a concept ga…	1
2	A Sustainable Urban Future Car-free cities …	1
3	Pioneering Sustainable Urban Living In an e…	1
4	The Path to Sustainable Urban Living In an …	1

[17]:
```
# renaming columns
external_testing_df = external_testing_df.rename(
    columns={"text": "Text", "generated": "Source"}
```

```
)
# Create the 'Length' column
external_testing_df["Length"] = external_testing_df["Text"].apply(lambda x:␣
 ↪len(str(x)))


# Create the 'Topic' column
external_testing_df["Topic"] = "external_test"


# Clean the 'Text' column to ensure all are strings
external_testing_df["Text"] = external_testing_df["Text"].astype(str)


# Remove rows with NaN values in 'Text' column
external_testing_df.dropna(subset=["Text"], inplace=True)


external_testing_df.head()
```

[17]:

	Text	Source	Length	\
0	carfree cities have become a subject of increa…	1	4035	
1	car free cities carfree cities, a concept gain…	1	3713	
2	a sustainable urban future carfree cities are …	1	3781	
3	pioneering sustainable urban living in an era …	1	3693	
4	the path to sustainable urban living in an age…	1	3651	

	Topic
0	external_test
1	external_test
2	external_test
3	external_test
4	external_test

[41]:
```
for col in external_testing_df.columns:
    if col == "Source":
        external_testing_df[col] = external_testing_df[col].replace("human", 0)
```

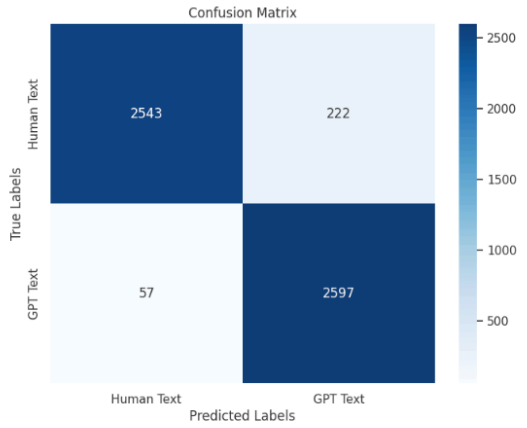
[42]:
```
# calling the function to clean the data
external_testing_df = lowercase_text(external_testing_df, "Text")
external_testing_df = remove_special_characters(external_testing_df, "Text")
external_testing_df["Text"] = external_testing_df["Text"].apply(clean_text)
external_testing_df = drop_nan_values(external_testing_df)
external_testing_df.to_csv("cleaned_data.csv", index=False)
```

[28]:
```
external_testing_df["Source"].describe()
```

[28]:
```
count    29145.000000
mean         0.399279
std          0.489759
min          0.000000
25%          0.000000
50%          0.000000
75%          1.000000
max          1.000000
Name: Source, dtype: float64
```

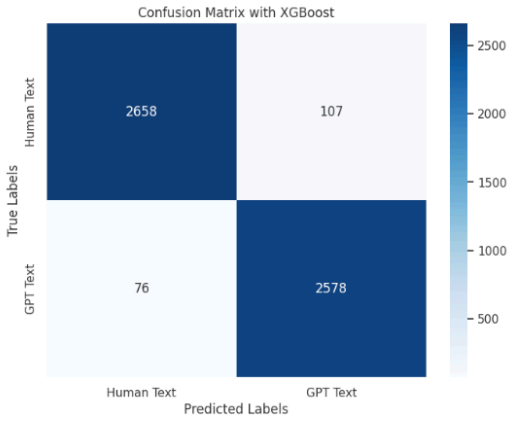
Section 7: Model Performance and Comparisons

Using 50-50 split Training and Testing data.




```
Accuracy: 0.9485144860675402
              precision    recall  f1-score   support

           0       0.98      0.92      0.95      2765
           1       0.92      0.98      0.95      2654

    accuracy                           0.95      5419
   macro avg       0.95      0.95      0.95      5419
weighted avg       0.95      0.95      0.95      5419
```

Figure 9. Results from the extratrees model using a 50-50 test split.

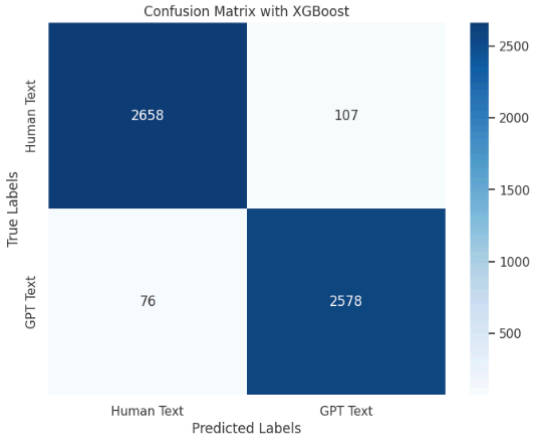



```
Accuracy: 0.9662299317217199
              precision    recall  f1-score   support

           0       0.97      0.96      0.97      2765
           1       0.96      0.97      0.97      2654

    accuracy                           0.97      5419
   macro avg       0.97      0.97      0.97      5419
weighted avg       0.97      0.97      0.97      5419
```

Figure 10. Results from the xgboost model using a 50-50 test split.

For our model comparison and performance metrics, we start with testing our data on a 50-50 split to ensure that our data does not overfit on the training data. Now though this is not a conventional 60:40/70:30 split, due to high performance in the first assignment(98% accuracy), I took this approach with the intention of mitigation overfitting. Despite adopting this strategy, our models continue to exhibit remarkably high performance, with the ExtraTrees model achieving a 94% accuracy rate and the XGBoost model reaching 96% accuracy. While these figures are marginally lower than those observed in the first pipeline, they still raise concerns. In an effort to address potential overfitting issues further, I plan to conduct a cross-validation training on the XGBoost model.

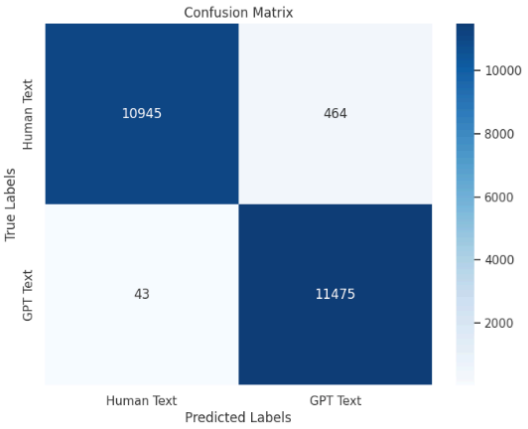
Using cross-validation




```
Cross-validation Accuracy (Mean): 0.973
Cross-validation Accuracy (Standard Deviation): 0.003
Accuracy: 0.9662299317217199
              precision    recall  f1-score   support

           0       0.97      0.96      0.97      2765
           1       0.96      0.97      0.97      2654

    accuracy                           0.97      5419
   macro avg       0.97      0.97      0.97      5419
weighted avg       0.97      0.97      0.97      5419
```

As we observe from the cross-validation, our model performance is not any better(with regards to overfitting). This makes me question whether our dataset is still very small and could be the potential reason that our model still seems to overfit regardless of showing impressive validation results with the test data both for precision and recall. To investigate this, I ingested more external human data as we already had a substantial amount of chatGPT-generated text.

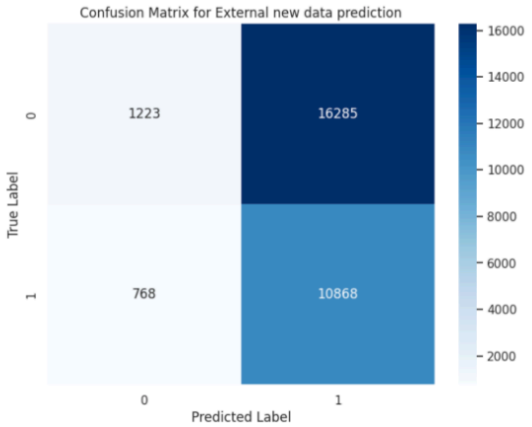
Using an even larger dataset



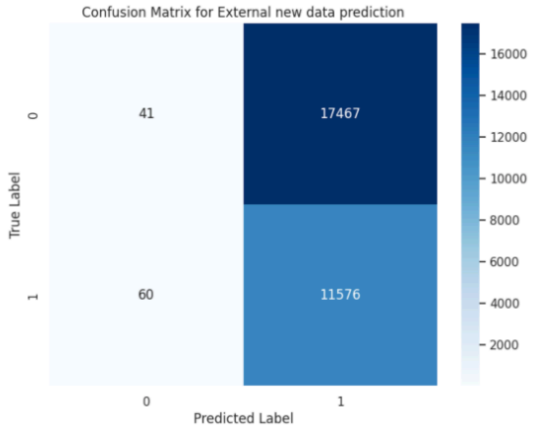

```
Accuracy: 0.9778863348889955
              precision    recall  f1-score   support

           0       1.00      0.96      0.98     11409
           1       0.96      1.00      0.98     11518

    accuracy                           0.98     22927
   macro avg       0.98      0.98      0.98     22927
weighted avg       0.98      0.98      0.98     22927
```

While still shocking, our model performs even better despite having more than ten thousand data points for each class. This made me question the integrity of the external datasets and my training architecture. Though my pipeline takes the appropriate steps in text processing, the only bulletproof strategy to ensure that our model was not overfitting was to test the models on new data that it had not seen before.

External Dataset Prooftesting



Accuracy: 0.41487098545155093
Model: xgBoost model



Accuracy:0.39860691737578
Model: Extra trees model

Though I posit that the model may have overfit, I also fear that most of the external data tended to be sourced from domain-specific fields, such as only from blog posts or Wikipedia pages, making the models create domain-specific patterns that are associated with GPT/Human text. As a consequence, when presented with external data, the model performs worse due to the already biased patterns it had learned.

Conclusion

In conclusion, creating generalizable models to distinguish between human vs chatGPT accurately text is quite tricky, as one runs into domain overfitting, mainly because getting extremely varied data from different domains can be difficult. Additionally, when working with human vs GPT data, though GPT is more consistent with its writing as the domain changes, the style equally changes, making it difficult to find generalizable patterns. This can also be said for human data, as no two humans write the same, and finding generalizable patterns when we bundle all human data is equally a daunting task.

Consequently, we can do a somewhat decent classification of human vs. GPT-generated text, especially if the human text is from the same human and we have some domain specificity. However, when we use new external data from a different domain and humans, our models seem to perform extremely badly due to the reasons stated above.

Finally, though our models tended to overfit and performed very badly with external data, this is not to say that this is an impossible task to solve but rather a difficult one to get right.

References

Geurts, P., Ernst, D., & Wehenkel, L. (2006). Extremely randomized trees. *Machine Learning*, *63*, 3–42. https://doi.org/10.1007/s10994-006-6226-1

Gradient boosting and xgboost in machine learning: Easy explanation for data science interviews. (n.d.). *www.youtube.com*. Retrieved March 25, 2024, from https://www.youtube.com/watch?v=yw-E___nDkKU&t=309s

Islam, N., Sutradhar, D., Noor, H., Tasnim Raya, J., Tabassum Maisha, M., & Farid, D. (n.d.). Distinguishing human generated text from chatgpt generated text using machine learning. Retrieved March 25, 2024, from https://arxiv.org/pdf/2306.01761v1.pdf

Nlpaug.augmenter.word.back$_t$*ranslation—nlpaug*1.1.11*documentation*. (n.d.). *nlpaug.readthedocs.io*. Retrieved March 23, 2024, from https://nlpaug.readthedocs.io/en/latest/augmenter/word/back_translation.html

Nlpaug.augmenter.word.context$_w$*ord*$_e$*mbs—nlpaug*1.1.11*documentation*. (n.d.). *nlpaug.readthedocs.io*. Retrieved March 23, 2024, from https://nlpaug.readthedocs.io/en/latest/augmenter/word/context_word_embs.html

Nlpaug.augmenter.word.synonym — nlpaug 1.1.11 documentation. (n.d.). *nlpaug.readthedocs.io*. https://nlpaug.readthedocs.io/en/latest/augmenter/word/synonym.html

Pandya, H. (2021, July). Textgenie - augmenting your text dataset with just 2 lines of code! *Medium*. Retrieved March 23, 2024, from https://towardsdatascience.com/textgenie-augmenting-your-text-dataset-with-just-2-lines-of-code-23ce883a0715

Text augmentation to boost nlp model performance (part 2/3). (n.d.). *www.youtube.com*. Retrieved March 23, 2024, from https://www.youtube.com/watch?v=URLqjenW-2M

Yan, S. (2017, November). Understanding lstm and its diagrams. *Medium*. https://blog.mlreview.com/understanding-lstm-and-its-diagrams-37e2f46f1714