Model

Minerva University

CS156

Model

Contents

Executive Summary

In our project, we developed a nuanced text classification model to distinguish between human-generated text and text produced by ChatGPT. Initially, we scripted a process to load relevant data from Google Docs, which was then structured into a DataFrame for preprocessing. Utilizing natural language processing libraries such as NLTK, SpaCy, and regular expressions, we cleaned the data, setting the stage for insightful feature extraction. Our approach hinged on identifying distinct patterns and linguistic cues unique to ChatGPT responses, such as the structure and prescriptive language used, to inform our feature selection.

Subsequent data analysis on the human and GPT datasets guided our normalization strategy, ensuring that our model could learn from balanced and representative data. Adhering to a simple-to-complex model selection philosophy, we commenced with logistic regression, achieving suspiciously high performance metrics that prompted a deeper investigation into potential overfitting or data mismatches.

To refine our understanding and potentially capture more complex patterns autonomously, we transitioned to employing BERT, a more sophisticated model capable of feature self-identification. Despite the shift, the high performance persisted, underscoring the need for further examination of our data and feature engineering approach.

Concluding, our findings highlight the need for a more expansive data representation and refined feature discernment to enhance the model's applicability in real-world scenarios. Future recommendations focus on enlarging the dataset to encompass a wider array of linguistic variations, which is crucial for improving the model's ability to accurately classify text origins. Additionally, reconsidering the solution philosophy from merely classifying text as human or GPT-generated to estimating the probability of text being generated by either source might offer a more nuanced understanding of the data. This shift towards a probabilistic estimation approach allows for a more flexible and informative analysis, accommodating the complexities and subtleties inherent in natural language, and reflecting the degree of confidence in the model's predictions.

Section 1: Introduction

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.

Exploratory Question

Is it feasible to develop a machine learning model that accurately differentiates texts written by a specific individual (me) from those generated by ChatGPT?

Data Collection

In an attempt to solve the exploratory question, I use a Python script that gets chunks of text from a Google doc and adds it to a CSV file with the following columns; - Text - Source - human or ChatGPT - Length - character length - Topic My data collection process involved going back to all the text I had written pre-chatGPT. This included all my college essay applications, assignments, emails, and telegram chats with friends. I then put all this data into a single Google document and then passed it to the script to turn it into chunks for analysis with their respective label.

For chatGPT text, I went through all my previous chatGPT logs and pasted the responses into a Google doc then loaded it into my script for chunking. At this point, it is important to note that most of my ChatGPT responses seem to be in academic domains(specifically computer science,economics, and finance), and this might lead to some inherent bias in our training and test

data.

Sampling

For my personal text, I did not use a random sampling method but rather gathered my personal text data from Google Drive, beginning with the earliest backup until just before ChatGPT's introduction to avoid corruption with some of my text that might include GPT excerpts. On the other hand for chatGPT responses I randomly selected responses and only excluded.

Text Chunking process

To ensure that the chunks we input into our CSV are coherent sentences and hold adequate contextual information, our script leverages the Natural Language Toolkit (NLTK) for precise sentence tokenization. Specifically, we leverage the **nltk.tokenize.sent_tokenize** because it is adept at identifying the natural linguistic boundaries within the text, effectively separating our text into individual sentences chunked into paragraphs.

[ ]:
```
# 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")
```

Section 2: Loading the data

[ ]:
```
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.")
```

[256]:
```
# converting the data to a dataframe.
human_df = pd.read_csv("output.csv")
gpt_df = pd.read_csv("gpt_output.csv")



# comibining the data
master_data = pd.concat([human_df, gpt_df], ignore_index=True)
master_data.head()
```

[256]:

	Text	Source	Length	Topic
0	Writing Journal Entry 6 After this class, writ...	human	1892	journal
1	However, what makes the solo so captivating is...	human	1925	journal
2	Furthermore, the guitar being a limited editio...	human	1962	journal
3	Observing Kawara's art pieces, the concept of ...	human	1422	journal
4	Journal Entry 8 Choose one HC from each of the...	human	1986	journal

Section 3: Data Processing & Feature Engineering

Data Processing

In our data cleaning process, we implement a sequence of tailored functions to refine the dataset, ensuring it's primed for effective analysis and model training. The essence of this cleaning lies in enhancing data quality and improving our model's focus.

First, we convert our "Source(label)" column to binary representation with 0 representing texts written by humans and 1 text written by chatGPT. This ensures that the dataset is

compatible with machine learning algorithms requiring numerical input. This conversion simplifies the data preparation process and facilitates the model's ability to perform binary classification tasks efficiently.

Second, we convert all our text to lowercase letters to standardize the text data, improving the model's focus. This standardization ensures that the algorithm treats words like "Hello," "HELLO," and "hello" as the same word, thereby reducing the complexity of the text data and the feature space. With the same intention of standardizing our data, we also remove any extra spaces, special characters, and nonvalues to reduce our feature space further, enhance tokenization accuracy, and optimize the model's processing time.

Finally, since we are not particularly interested in sentiment analysis in our text corpus, we can be confident that this process will not reduce our model's accuracy.

[196]:
```
# cleaning the data
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):
```

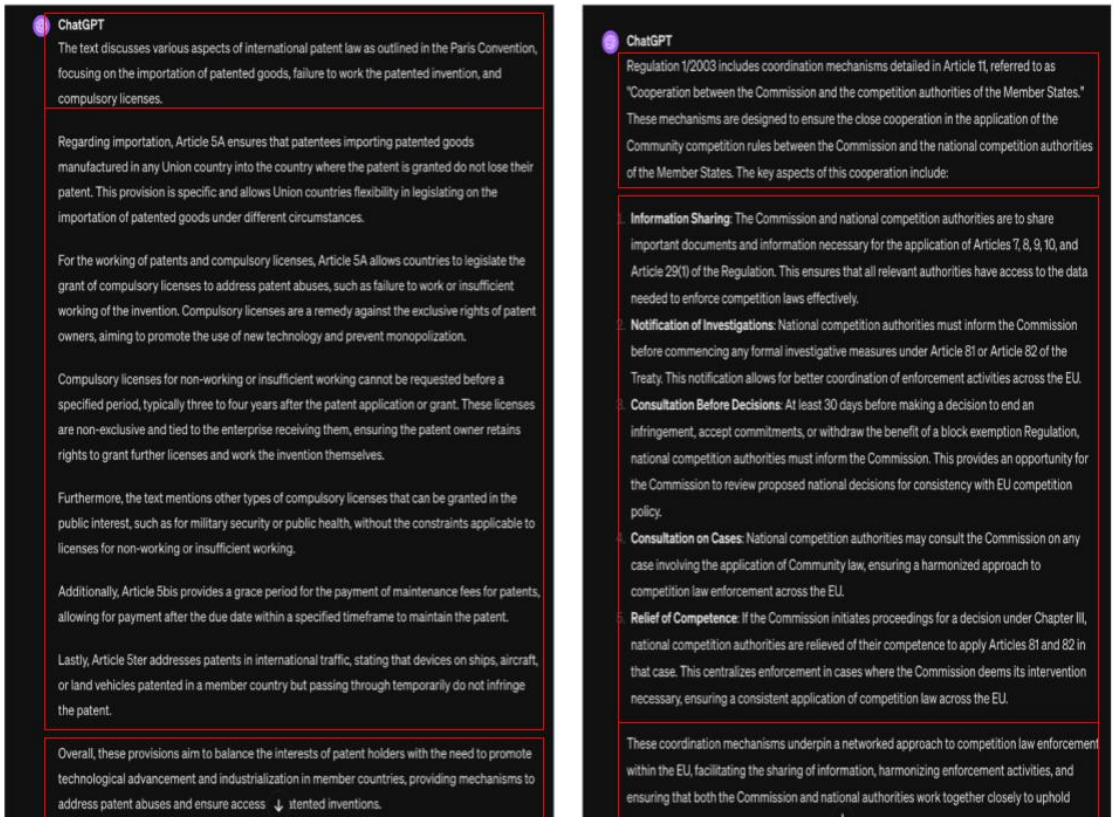
```
    df[text_col] = df[text_col].apply(lambda x: re.sub(r"[^a-z0-9\s,.?!]", "",␣
 ↪x)) # Remove special characters
    return df



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
```

[197]:
```
# 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)
```

Feature Extraction

When selecting what features to use to distinguish between chatGPt and human text, we asked ourselves how humans are able to differentiate between human and chatGPT text. However, before stating our findings, it is important to note that even humans are sometimes unable to make distinctions; however, we believe that we can use the heuristics that humans employ as a starting point in building our model.

Under the assumption that humans do not explicitly instruct the LLM to sound like a human, we noticed the following patterns. - ChatGPT seems to have a preference for a particular answer structure, as shown in the image below.



From the image above we can see that a majority of GPT responses(at least those from my dataset) seem to have a opening segment, body and conclusion summarizing the text altogether. Now though humans also tend to write in this manner, humans in most use case would not directly take the entirety of the text as it is especially if they want to incoroprate it into a larger body of work. However, though the structure and overly formal style of writing can

provide hints into chatGPT generated text it is not enough to make a distinction.

Nonetheless,if we observe the images below, we can also see another pattern where most responses start with phrases such as certainly! and ofcourse! followed by prescriptive language guiding the user through the provided information or steps to resolve their query. Furthermore, sometimes even for the simplest of question chatGPT tends to be unusually formal even in contexts that are meant to be conversational or casual. This factors coupled with its lack of personalized touch formed the foundation for our feature extraction.

With that in mind here are the features we extract from our text to classify if it is written by me or chatGPT. -

SyntacticComplexity: This feature is calculated by determining the depth of the parse tree for each sentence in the text. A parse tree represents the syntactic structure of a sentence according to a given grammar. In order to calculate this complexity, we convert the sentences into tokens and for each token we traverse the tree from each token to the root, counting the steps (edges) needed to reach the topmost node (the root). This process involves iterating over each token and moving up the tree through its parents (using the .head property in spaCy's token object) until the token's head is itself, indicating it's the root. The maximum depth encountered across all tokens within a sentence gives the parse tree's overall depth. This number reflects the sentence's syntactic complexity, with higher values indicating more complex structures.

Instructive Language:

- **Modal Verbs:** The presence of modal verbs (e.g., can, could, may, might, must, shall, should, will, would) is identified in the text. We do this by leveraging the spaCy library which can accurately identify these modal verbs based on their grammatical role. We concentrate on modal verbs because they are essential for expressing degrees of certainty, ability, permission, and obligation. Based on the idea that chatGPT can hallucinate and always sounds very certain about what it says, we hypothesize that GPT responses would have a higher frequency and context of modal verb usage, providing a heuristic to distinguish it from human-generated text.

- **Imperative Mood:** This feature identifies sentences that likely contain an imperative structure, which means they are commands or requests. The heuristic checks for verbs in the base form that are either the first token in a sentence or are preceded by a conjunction or preposition, suggesting an order or directive. The use of imperative mood could signal instructional or directive language, which might vary between humans and chatGPT in terms of frequency and context. This is based on the aforementioned pattern noted that GPT responses tend to be more directive.

[283]:
```
# sentence complexity feature extraction
!python -m spacy download en_core_web_sm
import spacy
nlp = spacy.load("en_core_web_sm")
def parse_tree_depth(sentence):
    """
    Calculate the depth of a parse tree for a given sentence.
    """
    depths = []
    for token in sentence:
        # Depth of current token
        depth = 0
        while token.head != token:
            depth += 1
            token = token.head
        depths.append(depth)
    return max(depths) if depths else 0



def average_parse_tree_depth(text):
    """
```

```
    Calculate the average depth of parse trees for all sentences in the text.
    """
    doc = nlp(text)
    depths = [parse_tree_depth(sent) for sent in doc.sents]
    return sum(depths) / len(depths) if depths else 0


master_data["Syntactic_Complexity"] = master_data["Text"].apply(
    average_parse_tree_depth
)
```

```
Collecting en-core-web-sm==3.7.1
  Downloading https://github.com/explosion/spacy-
models/releases/download/en_core_web_sm-3.7.1/en_core_web_sm-3.7.1-py3-none-
any.whl (12.8 MB)
     12.8/12.8 MB

 Download and installation successful
You can now load the package via spacy.load('en_core_web_sm')
```

[199]:
```
def contains_modal_verb(sentence):
    """
    Check if the sentence contains a modal verb.
    """
    return any(token.tag_ == "MD" for token in sentence)


def is_imperative_verb(token):
    """
    Heuristic to identify imperative verbs. Checks if the verb is in base form
```

```
    and it's the first token or preceded by a conjunction/preposition.
    """
    return (token.tag_ == "VB" and
            (token.head == token or token.head.pos_ in ["CONJ", "ADP"]))

def contains_imperative(sentence):
    """
    Check if the sentence likely contains an imperative structure.
    """
    for token in sentence:
        if is_imperative_verb(token):
            return True
    return False

def analyze_instructive_language(text):
    """
    Analyze the text for instructive language by identifying modal verbs and
    imperative mood structures.
    """
    doc = nlp(text)
    sentences_with_modal = sum(contains_modal_verb(sent) for sent in doc.sents)
    sentences_with_imperative = sum(contains_imperative(sent) for sent in doc.
↪sents)

    return {
        "total_sentences": len(list(doc.sents)),
        "sentences_with_modal": sentences_with_modal,
        "sentences_with_imperative": sentences_with_imperative,
```

```
    }


master_data["Instructive_Language"] = master_data["Text"].
 ↪apply(analyze_instructive_language)
```

Data Normalization

Considering that we combined our chatGPT responses and Human responses into a single dataframe we run a short analysis on the measures of central tendancy to identify skewness,outliers or major differences in the distribution of the data between the two soures of our data.

[291]:
```
print("This is an analysis of the human responses length.")
print(human_df["Length"].describe())
print("--------------------------------------")
print("This is an analysis of the GPT responses length.")
gpt_df["Length"].describe()
```

```
This is an analysis of the human responses length.
count    2047.000000
mean      152.468002
std       348.237420
min         1.000000
25%        13.000000
50%        28.000000
75%        78.500000
max      3737.000000
Name: Length, dtype: float64
--------------------------------------
This is an analysis of the GPT responses length.
```

```
[291]: count    2048.000000
       mean     1556.297363
       std       354.339503
       min       584.000000
       25%      1389.000000
       50%      1442.000000
       75%      1485.000000
       max      2501.000000
       Name: Length, dtype: float64
```

Running some analysis of the lengths of each data point in each data frame, we observe significant disparities, particularly in the average lengths—152 for human responses and 1556 for GPT responses. The maximum length also shows a stark contrast, with GPT's max at 2501 compared to human's 3737, highlighting the broader range in human responses. Considering our features, such as imperative and modal cues, are dependent on the frequency of certain words, the significant disparity on average between the human and GPT responses could bias our model classifications; thus, we normalize our data to account for this.

How do we normalize? To normalize our data, we use a technique that adjusts our feature values so they're on a similar scale. This is crucial because in machine learning, differences in scale can mess up how the model learns, potentially giving too much weight to one feature over another just because of the way the numbers look, not because that feature is actually more important.

We achieve this by using `StandardScaler` from Scikit-learn. This tool standardizes features by removing the mean and scaling to unit variance. The standardization process involves the following steps:

1. Calculating the mean of each feature:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Where x_i represents each value of the feature, and N is the number of values in the feature.

2. Calculating the standard deviation of each feature to understand the spread of the values in the feature around the mean:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

3. For each value x_i in the feature, subtract the mean μ of the feature and divide by the standard deviation σ of the feature. The result is z_i, the standardized value:

$$z_i = \frac{x_i - \mu}{\sigma}$$

In the code below, we specifically target features like "Length" and "Syntactic_Complexity" because we've identified that their scales could throw off our analysis or model training.

Finally, we also normalize our data by taking the ratios of our features(modal & imperative counts) in order to make comparisons of these linguistic features across texts of varying lengths

Why do we normalize? By normalizing, we're making sure every feature plays fair in influencing our machine learning models, ensuring more reliable and unbiased outcomes.

[201]:
```
import pandas as pd
from sklearn.preprocessing import StandardScaler



def extract_instructive_language_features(data):
    # Extracts and calculates features from 'Instructive_Language'
    data["total_sentences"] = data["Instructive_Language"].apply(
        lambda x: x["total_sentences"]
    )
    data["sentences_with_modal"] = data["Instructive_Language"].apply(
        lambda x: x["sentences_with_modal"]
    )
```

```
    data["sentences_with_imperative"] = data["Instructive_Language"].apply(
        lambda x: x["sentences_with_imperative"]
    )
    data["modal_sentence_ratio"] = (
        data["sentences_with_modal"] / data["total_sentences"]
    )
    data["imperative_sentence_ratio"] = (
        data["sentences_with_imperative"] / data["total_sentences"]
    )
    return data



def normalize_features(data, feature_names):
    # Normalizes specified features using StandardScaler
    scaler = StandardScaler()
    scaled_features = scaler.fit_transform(data[feature_names])
    for i, feature_name in enumerate(feature_names):
        data[feature_name] = scaled_features[:, i]
    return data


master_data = extract_instructive_language_features(master_data)



feature_names_to_normalize = ["Length", "Syntactic_Complexity"]
master_data = normalize_features(
    master_data, feature_names_to_normalize
)
```

Section 4: Task Discussion

Given the goal of differentiating between ChatGPT-generated text and human-generated text, our task is framed as a binary classification problem.Our strategy hinges on leveraging unique linguistic features, particularly focusing on syntactic complexity, modal verb usage, imperative sentence frequency, and their respective ratios to total sentences.We hypothesize that to some degree these features can capture distinctive patterns that could differentiate AI from human writing styles. In our initial analysis, we had included character length as a feature however though it was normalized it seemed to greatly bias our model thus we excluded it as a feature.

Data Splitting

We use the **train_test_split** function from scikit-learn, a trusted library in the machine learning community, to ensure a methodical and reproducible division of our dataset.

Specifically, we allocate 70% of our dataset to the training set and the remaining 30% to the test set. This ratio is carefully chosen to provide our model with a substantial amount of data for learning, while still reserving a significant portion for testing its performance on data it hasn't seen before. The random_state parameter is set to a fixed value, which guarantees that our split is reproducible; anyone rerunning our code will obtain the same training and test sets, ensuring consistency in evaluation and comparison of results.

[236]:
```
from sklearn.model_selection import train_test_split



regression_data = master_data.dropna()
# Define your features and target variable
X = regression_data[
    [
        "Syntactic_Complexity",
        "sentences_with_modal",
        "sentences_with_imperative",
```

```
        "modal_sentence_ratio",
        "imperative_sentence_ratio",
    ]
]
y = regression_data["Source"]


# Split the data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
```

Section 5: Model Selection

Model Selection Philosophy

When approaching the task of model selection, our philosophy is grounded in the principle of starting simple and iterating towards complexity as needed. This approach allows us to initially employ models that are easier to understand, interpret, and implement. It also helps in quickly establishing a baseline performance for our problem. The decision to transition to more complex models is driven by an assessment of bias (underfitting) and variance (overfitting) observed in our initial models. If a simple model exhibits high bias, indicating that it is too simplistic to capture the underlying patterns in the data, we might consider a more complex model that can learn these patterns more effectively. Conversely, if our model exhibits high variance, we might seek to simplify our approach or employ techniques like regularization to mitigate overfitting.

Model 1:LogisticRegression

Logistic Regression in scikit-learn is implemented as a linear model for classification, not regression, despite its name. It is designed to model the probabilities of the possible outcomes of a single trial using a logistic function, making it suitable for binary, One-vs-Rest, or multinomial logistic regression with optional regularization.

Why Logistical Regression?

I opted for Logistic Regression due to its simplicity, aligning with our model selection philosophy that emphasizes starting with straightforward models. This choice is particularly pertinent for our binary classification challenge, where we need to distinguish between texts generated by humans and GPT. Logistic Regression not only fits our requirement for a binary classification but also serves as an excellent baseline model. This approach allows us to establish a performance benchmark, which is crucial for evaluating the effectiveness of more complex models that might be considered later. The model's ability to provide probabilities for each class further aids in understanding the confidence level behind each prediction, an essential aspect when assessing the origin of the text. Given these considerations, Logistic Regression was deemed the most fitting initial model for our analysis, balancing both efficiency and interpretability.

How it works

Mathematical Formulation

For the binary case, logistic regression predicts the probability

$$P(y_i = 1|X_i)$$

for the positive class as follows:

$$\hat{p}(X_i) = \frac{1}{1 + \exp(-X_i w - w_0)},$$

where X_i is the input feature vector, w is the coefficient vector, and w_0 is the intercept. In the context of our logistic regression model, the input feature vector X_i comprises various linguistic metrics such as Syntactic Complexity, sentences with modal verbs, sentences with imperative mood, modal sentence ratio, and imperative sentence ratio. These features are extracted from the `regression_data` dataset and are used to predict the 'Source' variable, which classifies text by its origin. The coefficient vector w represents the weights assigned by the model to each of these features to make predictions, and w_0

Cost Function

The cost function for logistic regression aims to find the parameter values for the coefficient vector w and intercept w_0 that minimize the discrepancy between the predicted probabilities $\hat{p}(X_i)$ and the actual outcomes y_i. It is expressed as:

$$\min_{w} C \sum_{i=1}^{n} [-y_i \log(\hat{p}(X_i)) - (1 - y_i) \log(1 - \hat{p}(X_i))] + r(w),$$

where:

- C is the inverse of regularization strength. A smaller value of C specifies stronger regularization, helping to prevent overfitting by penalizing large values of the coefficients.
- y_i represents the actual class label for the i^{th} observation, either 0 or 1.
- $\hat{p}(X_i)$ is the predicted probability of the i^{th} observation being in class 1, as calculated by the logistic function.
- $r(w)$ denotes the regularization term, which can take various forms:
  - For $r(w) = 0$, no regularization is applied.
  - For $r(w) = \|w\|_1$, L1 regularization is applied, encouraging sparsity in the coefficient vector (i.e., making some coefficients zero).
  - For $r(w) = \frac{1}{2}\|w\|_2^2$, L2 regularization is applied, encouraging smaller coefficients evenly (i.e., close to zero but not exactly zero).
  - For a combination of L1 and L2, known as Elastic-Net, $r(w)$ is a linear combination of both, controlled by a mixing parameter.

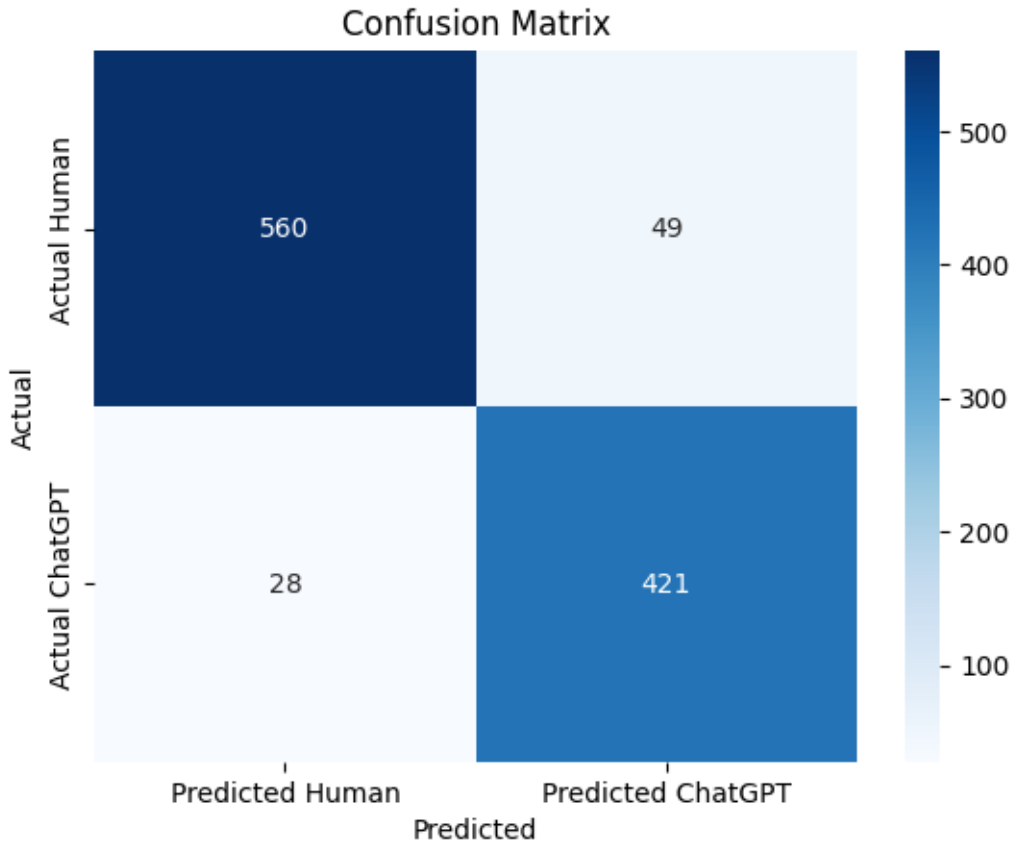
This cost function is optimized through gradient descent or other optimization algorithms to find the best values of w and w_0 that minimize the cost. The regularization term is crucial for controlling the complexity of the model, balancing the trade-off between fitting the training data well and keeping the model simple enough to generalize to new data.

Finally, in our logistic regression model, the cost function is a pivotal component that guides the optimization process, aiming to find the best parameter values that minimize prediction errors on our data.

[242]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt


# Initialize the model
model = LogisticRegression()


# Train the model
model.fit(X_train, y_train)


# Predict on the test set
predictions = model.predict(X_test)


# Evaluate the model with classification report
print(classification_report(y_test, predictions))


# Generate and visualize the confusion matrix
cm = confusion_matrix(y_test, predictions)
sns.heatmap(
    cm,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=["Predicted Human", "Predicted ChatGPT"],
    yticklabels=["Actual Human", "Actual ChatGPT"],
)
plt.ylabel("Actual")
```

```
plt.xlabel("Predicted")
plt.title("Confusion Matrix")
plt.show()
```

	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



Model Evaluation

From the output of the model above we can make the following inferences about the evaluation of the data.

- **Precision:** For class 0 (*Human*), the precision is 0.95, meaning that 95% of instances predicted as *Human* were correct. For class 1 (*ChatGPT*), the precision is 0.90, so 90% of instances predicted as *ChatGPT* were correct.
- **Recall:** The recall for class 0 is 0.92, indicating that the model correctly identified 92% of all actual *Human* instances. Class 1 has a recall of 0.94, meaning it correctly identified 94% of all *ChatGPT* instances.
- **F1-Score:** The F1-scores are high for both classes (0.935 for *Human* and 0.92 for *ChatGPT*), suggesting a good balance between precision and recall.

Looking at the confusion matrix, we see a clearer picture of the classification with the majority of predictions aligning with the actual labels.The matrix shows that the model is more likely to correctly identify both 'Human' and 'ChatGPT' texts, with a higher number of true positives in both categories.

However, the exceptionally high precision and recall metrics are quite alarming because in practice, it's rare for a classification model to achieve near-perfect performance unless the data and the problem are very well-understood and the model is exceptionally well-tuned. Additionally, knowing that OpenAI pulled down their GPT/Human detector due to being inaccurate further fuels our suspicion that something is amiss in our model.

Based on these metrics, we assume the following as potential causes of such results:

- **Overfitting:** The model may have learned the training data too well, capturing idiosyncrasies that do not generalize to unseen data. This often happens if the model is too complex relative to the simplicity of the task.
- **Data Leakage:** There may have been an inadvertent inclusion of information in the training data that would not be available in actual prediction scenarios, allowing the model to cheat by learning from features that won't be present in real-world data.

- **Class Imbalance:** If the dataset had an uneven distribution of classes and this wasn't accounted for during model training or evaluation, the model's performance metrics might be skewed, showing artificially high scores.

To address this, we run the same model but this time we use cross-validation with the intention of providing a more accurate estimate of the model's performance on unseen data. This should mitigate the risk of overfitting, as it involves dividing the data into multiple folds and ensuring that the model is tested against different subsets.Additionally, cross-validation can help detect issues of data leakage or class imbalance, as consistent performance across all folds can indicate that the model's high accuracy is not due to these potential problems.

[238]:
```
from joblib import dump, load


# Assuming 'model' is your trained LogisticRegression model
dump(model, "logistic_regression_model.joblib")
```

[238]:
```
['logistic_regression_model.joblib']
```

[287]:
```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression


# Choose your model
model = LogisticRegression()


# Define the scoring metrics you want to use
scoring_metrics = ["accuracy", "precision", "recall", "f1", "roc_auc"]


# Perform cross-validation and store the results
cv_scores_dict = {}
for metric in scoring_metrics:
    cv_scores = cross_val_score(model, X_train, y_train, cv=5, scoring=metric)
```

```
    cv_scores_dict[metric] = np.mean(cv_scores)
    print("Average 5-Fold CV Score ({}): {:.2f}".format(metric,␣
↪cv_scores_dict[metric]))
```

```
Average 5-Fold CV Score (accuracy): 0.93
Average 5-Fold CV Score (precision): 0.91
Average 5-Fold CV Score (recall): 0.95
Average 5-Fold CV Score (f1): 0.93
Average 5-Fold CV Score (roc_auc): 0.98
```

Cross Validation Evaluation

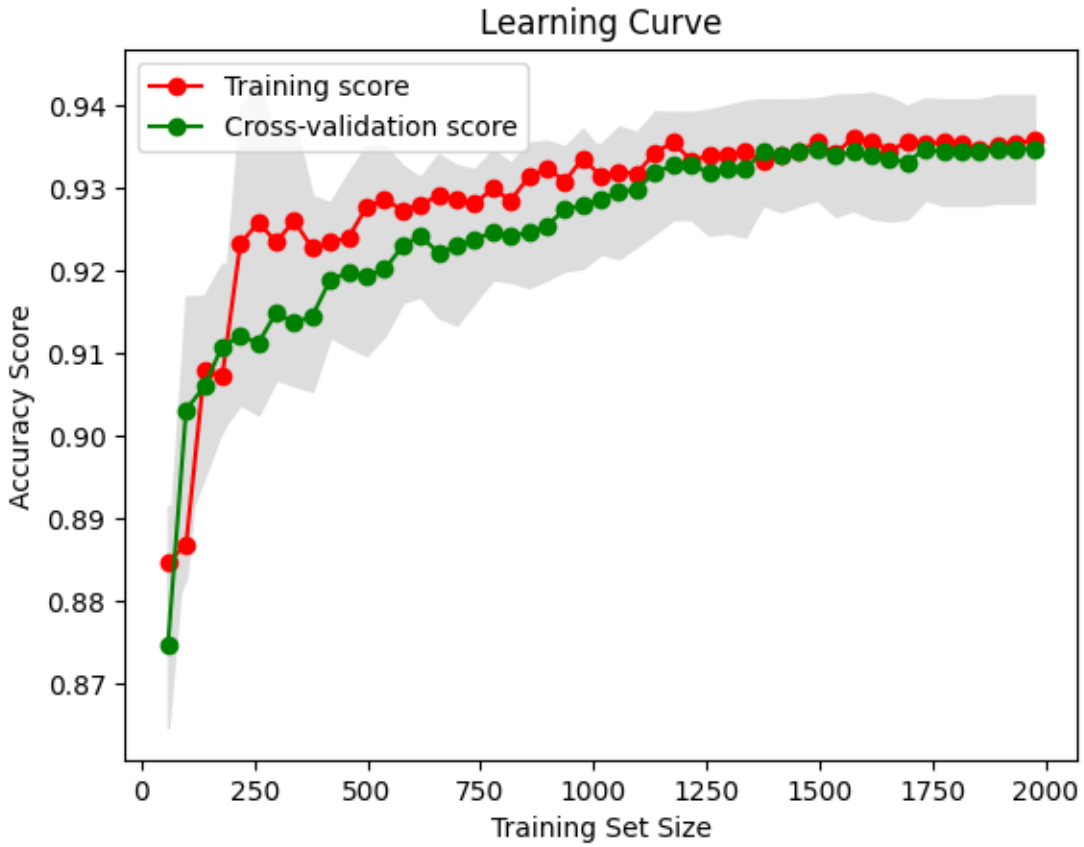
Given the absence of overfitting or data leakage as indicated by our cross-validation metrics, this prompts us to delve deeper into the representation of our data. The high performance across accuracy, precision, recall, F1, and ROC_AUC scores suggests that the model is effectively capturing the underlying patterns within the dataset. However, it also raises questions about the complexity and diversity of the data itself.

To further assess the robustness and generalizability of our model, we will first examine the model's learning curve. This will offer insights into how well the model is learning from the data over iterations of training, including whether it benefits from more data or if it plateaus, which could indicate issues such as overfitting or underfitting. This output can be seen below.

[249]:
```
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve


train_sizes, train_scores, validation_scores = learning_curve(
    estimator=model,
    X=X_train,
    y=y_train,
    train_sizes=np.linspace(0.01, 1.0, 50),
    cv=5,
```

```
    n_jobs=-1,
)

# Calculate mean and standard deviation for training set scores
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)

# Calculate mean and standard deviation for validation set scores
validation_mean = np.mean(validation_scores, axis=1)
validation_std = np.std(validation_scores, axis=1)

# Plot the learning curves
plt.fill_between(
    train_sizes, train_mean - train_std, train_mean + train_std, color="#DDDDDD"
)
plt.fill_between(
    train_sizes,
    validation_mean - validation_std,
    validation_mean + validation_std,
    color="#DDDDDD",
)

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

plt.title("Learning Curve")
plt.xlabel("Training Set Size")
```

```
plt.ylabel("Accuracy Score")

plt.legend(loc="best")

plt.show()
```



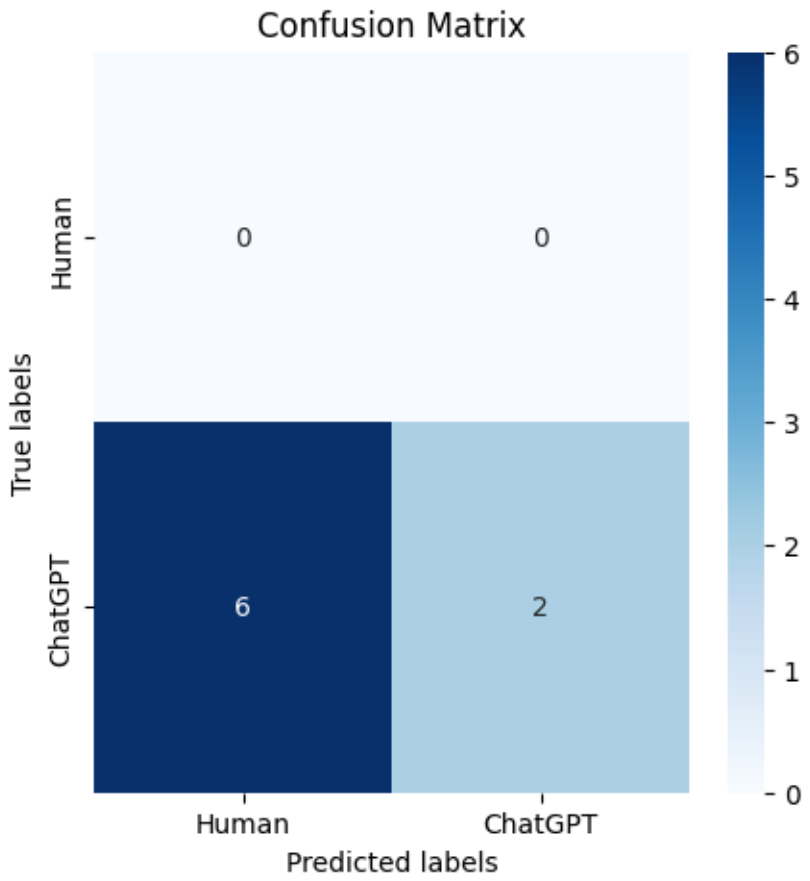
From the image above, it is interesting that even with a training set as small as 250, our model already has an accuracy of 91%. Given how difficult it is to distinguish between both chatGPT and human-generated text in the real world, we conclude that assuming the convergence between the 70/30 split and cross-validated reduces the probability that it could be a result of overfitting, turning our attention to whether the high accuracy could be as a result of lack of proper representation of the data. To test if this is true, we have a script below that takes in new external data, pre-processes it, and pipes it into the model, outputting a classification report as well as a confusion matrix.

[245]:
```
import pandas as pd
from joblib import load
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
)
import seaborn as sns
import matplotlib.pyplot as plt


def predict_new_data(file_path):
    # Load the new data
    new_data = pd.read_csv(file_path)

    # Apply transformations to new_data
    new_data = convert_source_labels(new_data, "Source")
    new_data = lowercase_text(new_data, "Text")
    new_data = remove_special_characters(new_data, "Text")
    new_data["Text"] = new_data["Text"].apply(clean_text)
    new_data = drop_nan_values(new_data)
    new_data["Syntactic_Complexity"] = new_data["Text"].apply(
        average_parse_tree_depth
    )
    new_data["Instructive_Language"] = new_data["Text"].apply(
        analyze_instructive_language
    )
```

```
new_data = extract_instructive_language_features(new_data)
new_data = normalize_features(new_data, feature_names_to_normalize)

# Define your features and target variable
X_new = new_data[
    [
        "Syntactic_Complexity",
        "sentences_with_modal",
        "sentences_with_imperative",
        "modal_sentence_ratio",
        "imperative_sentence_ratio",
    ]
]
y_new = new_data["Source"]

model = load("logistic_regression_model.joblib")
new_predictions = model.predict(X_new)

# Evaluate the model
# Now, evaluate the model's performance
accuracy = accuracy_score(y_new, new_predictions)
precision = precision_score(y_new, new_predictions)
recall = recall_score(y_new, new_predictions)
f1 = f1_score(y_new, new_predictions)

# Printing out the performance metrics
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
```

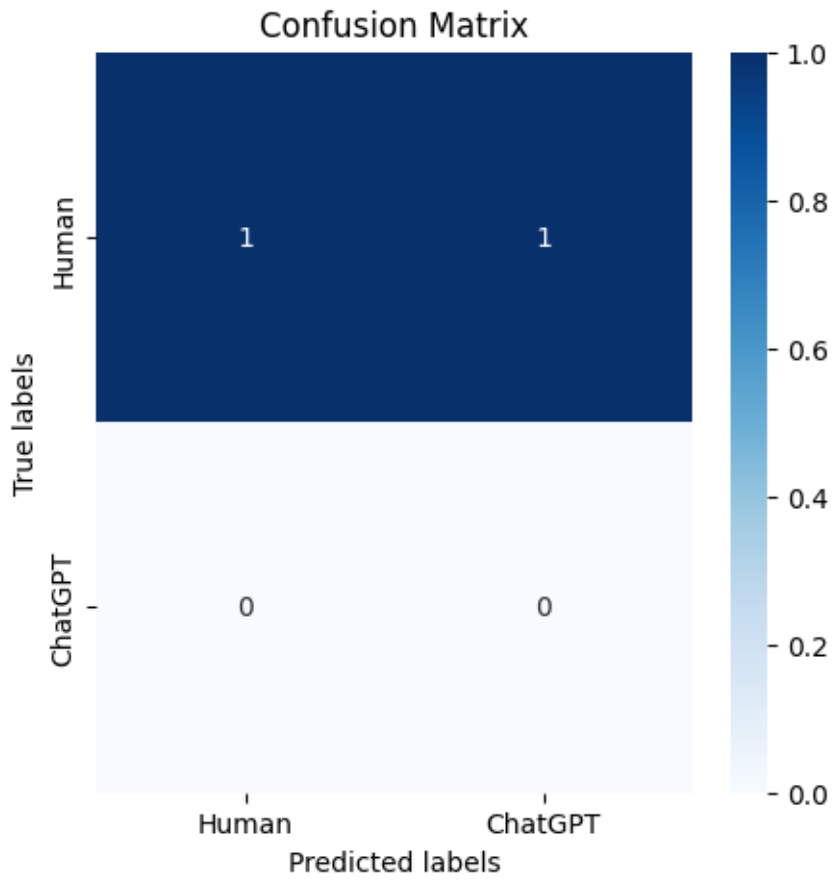
```
    print(f"Recall: {recall}")
    print(f"F1 Score: {f1}")



    # Generate and plot the confusion matrix
    conf_matrix = confusion_matrix(y_new, new_predictions)
    fig, ax = plt.subplots(figsize=(5, 5))
    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", ax=ax)
    ax.set_xlabel("Predicted labels")
    ax.set_ylabel("True labels")
    ax.set_title("Confusion Matrix")
    ax.xaxis.set_ticklabels(["Human", "ChatGPT"])
    ax.yaxis.set_ticklabels(["Human", "ChatGPT"])
    plt.show()

    return accuracy, report



print('This if for GPT generated reddit comments')
accuracy, report = predict_new_data("gpt_test.csv")
```

```
This if for gpt generated reddit comments
Accuracy: 0.25
Precision: 1.0
Recall: 0.25
F1 Score: 0.4
```



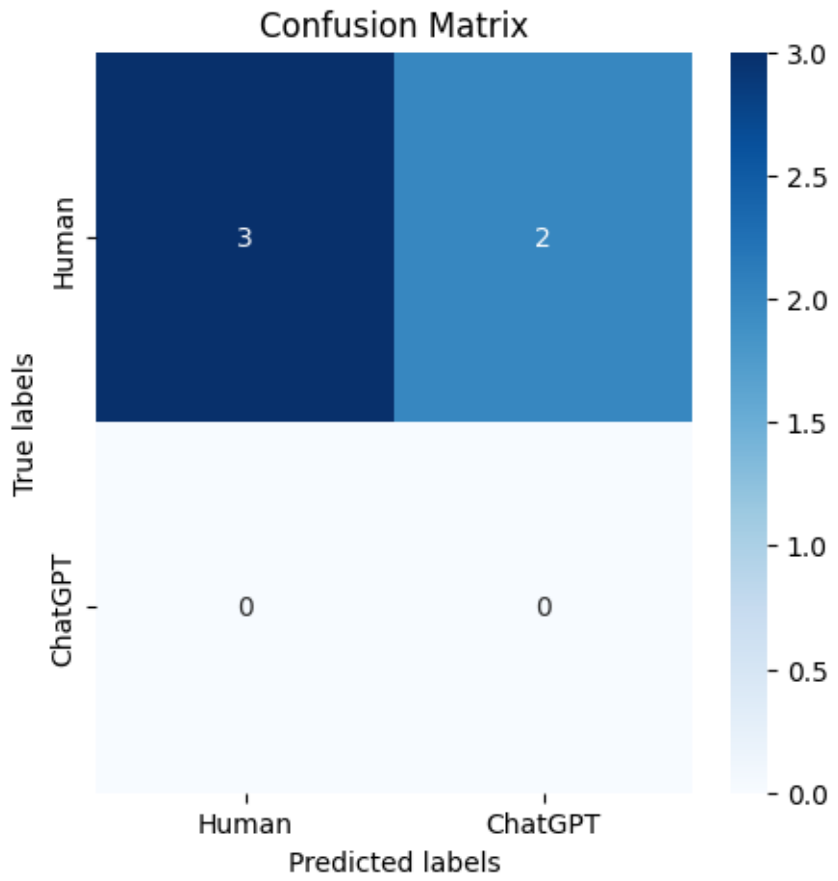
[288]:
```
print("This if for human generated reddit comments")

acc, report = predict_new_data("human_reddit.csv")
```

```
Accuracy: 0.5

Precision: 0.0

Recall: 0.0

F1 Score: 0.0
```



[290]:
```
print("This if for human generated social science(SS50) assignment")

acc, report = predict_new_data("sample_SS50_report.csv")
```

```
This if for human generated social science(SS50) assignment

Accuracy: 0.6

Precision: 0.0

Recall: 0.0

F1 Score: 0.0
```



Real World Test Analysis

From examining the samples of never-seen data above, we see that the model's accuracy significantly drops, ranging from as low as 0.25 to as high as 0.6, slightly better than chance. This raises the question: Why does the model perform so well on training and validation data but so poorly on external data? Some potential explanations for the discrepancy in model performance could include:

- **Data Distribution Mismatch:** The training and validation data might not represent the broader range of scenarios the model encounters with external data. This discrepancy suggests that the model has learned patterns specific to the training set that do not generalize well.
- **Overfitting:** The model might have overfitted to the noise or specific patterns in the

training data, making it perform exceptionally well on similar data but poorly on data that diverges from these patterns.

- **Feature Relevance:** The features used for training the model may not capture the essence of the problem space effectively when applied to external data. This can occur if the features are too specific to the training dataset or if critical predictors are missing.
- **Lack of Data Representation:** The training dataset may not encompass a sufficiently wide array of examples to cover all the variations the model might face in practical scenarios.

Given the possibility that our initial feature selection was too restrictive, we decided to pivot towards a model capable of autonomously discerning textual patterns without relying on manually extracted features. This approach aims to leverage the model's inherent ability to identify and learn from the intricacies and nuances of the text data directly, thereby potentially uncovering more complex relationships and patterns that our initial feature engineering might have missed.

Model Selection II :BERT (Bidirectional Encoder Representations from Transformers)

The chosen model for this task is based on BERT (Bidirectional Encoder Representations from Transformers), a state-of-the-art machine learning technique for natural language processing (NLP) tasks. BERT's architecture enables it to understand the context of words in a sentence by considering the words that come before and after, unlike traditional models that view each word in isolation.

How the model works. How the Model Works: - **Tokenization:** The first step involves using BERT's tokenizer to convert text data into a format that the model can understand. This process includes splitting the text into tokens (words or subwords), adding special tokens (like [CLS] for classification tasks and [SEP] for separating sentences), and converting these tokens into numerical IDs.

- **Encoding:** Each token is then passed through the BERT model, which encodes every token in the context of the sentence. The model consists of multiple layers of transformers, each capable of focusing on different parts of a sentence as it passes through the layers. The

output is a high-dimensional vector representing each token in the context of the entire text.

- **Classification:** For sequence classification tasks like ours, the encoded representation of the special [CLS] token (which is inserted at the beginning of each text) is typically used as the aggregate sequence representation for classification tasks. This representation is passed through additional layers (if any are specified) and a final classification layer to predict the class labels.

Loss Function: The loss function used in BERT for sequence classification tasks is typically Cross-Entropy Loss. In the context of our binary classification (distinguishing between 'Human' and 'GPT-generated' text), the Cross-Entropy Loss function measures the difference between the predicted probabilities assigned by the model and the actual class labels (0 or 1). Mathematically, it is defined as:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

In this expression: - N signifies the total number of samples in the dataset. - y_i denotes the true label of the i^{th} sample, where $y_i \in \{0, 1\}$. - $\hat{y}_i$ represents the model's predicted probability for the i^{th} sample being in class 1.

Role of AdamW Optimizer

The AdamW optimizer, a variant of the Adam optimizer with improved handling of weight decay regularization, is employed to minimize the loss function $\mathcal{L}$. It updates the model parameters θ (including weights w and bias b) in a way that navigates the multidimensional loss surface towards a global minimum. Mathematically, AdamW incorporates moment estimation and an adaptive learning rate for each parameter, facilitating efficient convergence:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

where: - θ_t and θ_{t+1} are the parameter values at iterations t and $t + 1$, respectively. - η is the step size or learning rate. - $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected estimates of the first and second moments of the gradients, respectively. - ϵ is a small scalar added to improve numerical stability.

The AdamW optimization strategy specifically modifies the way weight decay is applied, decoupling it from the gradient updates to enhance performance and generalization in deep learning models like BERT.

Implication in Contextual Representation Learning

By minimizing the Cross-Entropy Loss using AdamW, BERT learns to fine-tune its pre-trained contextual embeddings for the specific task of sequence classification. This process allows BERT to dynamically adjust its internal representations based on the complexity of the text data, capturing subtle linguistic nuances. The result is a model that can accurately classify sequences by effectively discerning patterns and relationships inherent in natural language, beyond what is possible through manual feature engineering alone.

[280]:
```
import numpy as np
import pandas as pd
import torch
from transformers import (
    BertTokenizer,
    BertForSequenceClassification,
    Trainer,
    TrainingArguments,
)
from torch.utils.data import Dataset
from sklearn.model_selection import KFold
import accelerate



# Function to merge dataframes containing human and GPT-generated texts into a␣
 ↪single dataframe
def load_data(human_df, gpt_df):
    return pd.concat([human_df, gpt_df], ignore_index=True)
```

```
# Function to tokenize the input texts using a specified tokenizer, also␣
 ↪converts labels into tensors
def tokenize_texts(tokenizer, texts, labels):
    # Tokenize all texts with truncation and padding
    encodings = tokenizer(texts, truncation=True, padding=True,␣
 ↪return_tensors="pt")
    # Convert labels into a tensor
    labels_tensor = torch.tensor(labels)
    return encodings, labels_tensor



# Custom dataset class to hold the tokenized text data and labels
class TextDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings  # Tokenized texts
        self.labels = labels  # Corresponding labels for each text

    def __getitem__(self, idx):
        # Retrieves an item by index and converts it into a format compatible␣
 ↪with PyTorch
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.
 ↪items()}
        item["labels"] = torch.tensor(self.labels[idx], dtype=torch.long)
        return item

    def __len__(self):
```

```
        # Returns the size of the dataset
        return len(self.labels)
```

[281]:
```
import numpy as np
import pandas as pd
import torch
from transformers import BertTokenizer, BertForSequenceClassification, AdamW
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, precision_recall_fscore_support
from tqdm import tqdm
from torch.utils.data import DataLoader



# Define a custom function to train and evaluate the model using K-Fold␣
↪cross-validation
def custom_train_eval_model(train_texts, train_labels, n_splits=3):
    # Initialize BERT tokenizer and model for sequence classification with 2␣
↪labels
    tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
    model = BertForSequenceClassification.from_pretrained(
        "bert-base-uncased", num_labels=2
    )

    # Set the device to GPU if available, else CPU
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    # Initialize K-Fold cross-validation
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
```

```
    # Initialize metrics storage
    all_accuracy = []
    all_precision = []
    all_recall = []
    all_f1 = []

    # Loop over each fold
    for fold, (train_idx, val_idx) in enumerate(kf.split(train_texts)):
        print(f"FOLD {fold}")
        print("--------------------------------")

        # Split the data for the current fold
        train_texts_fold = [
            str(text) for text in np.array(train_texts)[train_idx] if pd.
↪notnull(text)
        ]
        train_labels_fold = np.array(train_labels)[train_idx]
        val_texts_fold = [
            str(text) for text in np.array(train_texts)[val_idx] if pd.
↪notnull(text)
        ]
        val_labels_fold = np.array(train_labels)[val_idx]

        # Tokenize the texts for training and validation sets
        train_encodings, _ = tokenize_texts(
            tokenizer, train_texts_fold, train_labels_fold
        )
```

```
        val_encodings, _ = tokenize_texts(tokenizer, val_texts_fold,␣
↪val_labels_fold)

        # Create datasets from the tokenized texts and labels
        train_dataset = TextDataset(train_encodings, train_labels_fold)
        val_dataset = TextDataset(val_encodings, val_labels_fold)

        # Create DataLoaders for training and validation datasets
        train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
        val_loader = DataLoader(val_dataset, batch_size=8, shuffle=False)

        # Initialize the optimizer
        optimizer = AdamW(model.parameters(), lr=5e-5)

        # Training loop
        model.train()
        for epoch in range(1):  # Example: 3 training epochs
            for batch in tqdm(train_loader, desc=f"Training Epoch {epoch}"):
                optimizer.zero_grad()
                input_ids = batch["input_ids"].to(device)
                attention_mask = batch["attention_mask"].to(device)
                labels = batch["labels"].to(device)
                outputs = model(input_ids, attention_mask=attention_mask,␣
↪labels=labels)
                loss = outputs.loss
                loss.backward()
                optimizer.step()
```

```
        # Evaluation loop
        model.eval()
        all_preds = []
        all_true = []
        with torch.no_grad():
            for batch in tqdm(val_loader, desc=f"Evaluating, Fold {fold}"):
                input_ids = batch["input_ids"].to(device)
                attention_mask = batch["attention_mask"].to(device)
                labels = batch["labels"].to(device)
                outputs = model(input_ids, attention_mask=attention_mask)
                logits = outputs.logits
                preds = torch.argmax(logits, dim=1)
                all_preds.extend(preds.cpu().numpy())
                all_true.extend(labels.cpu().numpy())

        # Compute metrics for the fold
        accuracy = accuracy_score(all_true, all_preds)
        precision, recall, f1, _ = precision_recall_fscore_support(
            all_true, all_preds, average="binary"
        )

        # Store metrics
        all_accuracy.append(accuracy)
        all_precision.append(precision)
        all_recall.append(recall)
        all_f1.append(f1)

        print(f"Fold {fold} Accuracy: {accuracy}")
```

```
        print(f"Fold {fold} Precision: {precision}")
        print(f"Fold {fold} Recall: {recall}")
        print(f"Fold {fold} F1 Score: {f1}")

    # Print average metrics after all folds
    print(f"Average Accuracy: {np.mean(all_accuracy)}")
    print(f"Average Precision: {np.mean(all_precision)}")
    print(f"Average Recall: {np.mean(all_recall)}")
    print(f"Average F1 Score: {np.mean(all_f1)}")

    # Return the metrics as a dictionary
    return {
        "accuracy": all_accuracy,
        "precision": all_precision,
        "recall": all_recall,
        "f1": all_f1,
    }
```

[282]:
```
main_data = load_data(human_df, gpt_df)
main_data = lowercase_text(main_data, "Text")
main_data = remove_special_characters(main_data, "Text")
main_data["Text"] = main_data["Text"].apply(clean_text)
main_dataa = drop_nan_values(main_data)
train_texts = main_data["Text"].to_numpy()
train_labels = main_data["Source"].to_numpy()



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

```
# Call the custom training function
custom_train_eval_model(train_texts, train_labels, n_splits=3)
```

```
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at bert-base-uncased and are newly initialized:
['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.

FOLD 0
--------------------------------

Training Epoch 0:   0%|          | 0/342 [00:00<?, ?it/s]/var/folders/n1/7yb7bsv
906q1587trnj84z3w0000gn/T/ipykernel_64585/3936408271.py:32: UserWarning: To copy
construct from a tensor, it is recommended to use sourceTensor.clone().detach()
or sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
Training Epoch 0: 100%|| 342/342 [1:49:31<00:00, 19.21s/it]
Evaluating, Fold 0: 100%|| 171/171 [06:21<00:00,  2.23s/it]

Fold 0 Confusion Matrix:
[[713   1]
 [ 23 628]]
Fold 0 Accuracy: 0.9824175824175824
Fold 0 Precision: 0.9984101748807631
Fold 0 Recall: 0.9646697388632872
Fold 0 F1 Score: 0.98125
FOLD 1
--------------------------------
```

```
/Users/kyronnyoro/Code/A/venv/lib/python3.10/site-
packages/transformers/optimization.py:429: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
  warnings.warn(
Training Epoch 0:   0%|          | 0/342 [00:00<?, ?it/s]/var/folders/n1/7yb7bsv
906q1587trnj84z3w0000gn/T/ipykernel_64585/3936408271.py:32: UserWarning: To copy
construct from a tensor, it is recommended to use sourceTensor.clone().detach()
or sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
Training Epoch 0: 100%|| 342/342 [55:15<00:00,  9.70s/it]
Evaluating, Fold 1: 100%|| 171/171 [06:26<00:00,  2.26s/it]

Fold 1 Confusion Matrix:
[[663   4]
 [  0 698]]
Fold 1 Accuracy: 0.9970695970695971
Fold 1 Precision: 0.9943019943019943
Fold 1 Recall: 1.0
Fold 1 F1 Score: 0.9971428571428571
FOLD 2
--------------------------------

/Users/kyronnyoro/Code/A/venv/lib/python3.10/site-
packages/transformers/optimization.py:429: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
```

```
  warnings.warn(
Training Epoch 0:   0%|          | 0/342 [00:00<?, ?it/s]/var/folders/n1/7yb7bsv
906q1587trnj84z3w0000gn/T/ipykernel_64585/3936408271.py:32: UserWarning: To copy
construct from a tensor, it is recommended to use sourceTensor.clone().detach()
or sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
Training Epoch 0: 100%|| 342/342 [56:04<00:00,  9.84s/it]
Evaluating, Fold 2: 100%|| 171/171 [31:50<00:00, 11.17s/it]

Fold 2 Confusion Matrix:
[[664   2]
 [  1 698]]
Fold 2 Accuracy: 0.9978021978021978
Fold 2 Precision: 0.9971428571428571
Fold 2 Recall: 0.9985693848354793
Fold 2 F1 Score: 0.997855611150822
Average Accuracy: 0.9924297924297925
Average Precision: 0.9966183421085382
Average Recall: 0.9877463745662555
Average F1 Score: 0.9920828227645596
```

[282]:
```
{'accuracy': [0.9824175824175824, 0.9970695970695971, 0.9978021978021978],
 'precision': [0.9984101748807631, 0.9943019943019943, 0.9971428571428571],
 'recall': [0.9646697388632872, 1.0, 0.9985693848354793],
 'f1': [0.98125, 0.9971428571428571, 0.997855611150822]}
```

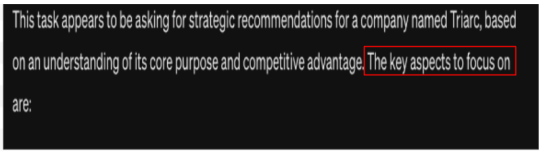
Discussion & Conclusion

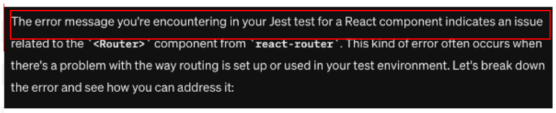
From the results above, we see that the BERT model has high accuracy, precision, and recall metrics, which are very similar to the results we got from our logistical regression. Though this seems to show that our features were effective, BERT's accuracy could be attributed to the fact that the data the model is trained on is not representative of real-world usage. Considering that the data used to train the model is based solely on my archive, we find that most of my chatGPT responses involved academic domains, whereas my personal text was more varied. Additionally, this also brings into question what exactly we categorize as chatGPT text. Considering the methodology used in collecting the dataset, where ChatGPT-generated texts were obtained without any modifications, it's important to acknowledge the potential discrepancy between these samples and real-world applications. In practical scenarios, humans often alter or edit the text generated by models like ChatGPT for various purposes, such as clarity, style, or specific content goals.

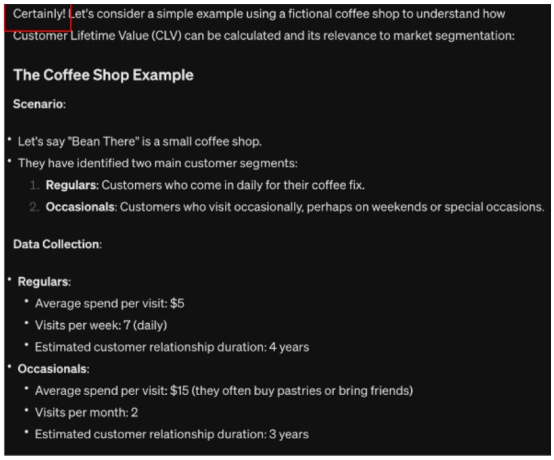
As a result, in the future, to account for these discrepancies, it would be beneficial to train our models on an extensive and well-representative dataset as well as changing our solution philosophy from a classification problem to a probability problem where we output the probability that some text is written by either a human or chatGPT. This is a more practical approach to solving the problem since, in cases where text is obviously chatGPT generated(which is what we want to focus on), our model can predict a higher probability while still having some room for error due to human intervention

References

Biswas, A. (2023, July 16). How to Build an AI Content Detector from Scratch with Python. Medium. https://levelup.gitconnected.com/how-to-build-an-ai-content-detector-from-scratch-with-python-c3cc432a2ee5Despoudis, T. (2023, September 21). How to Build an AI Text Detector Using Python. ActiveState. https://www.activestate.com/blog/ai-generated-text-detector/Kaufman, I. (2020, October 2). 5 Steps to Build an AI-based Fraud Detection System. Medium. https://towardsdatascience.com/5-steps-to-build-an-ai-based-fraud-detection-system-3f2411c02bbe


This task appears to be asking for strategic recommendations for a company named Triarc, based on an understanding of its core purpose and competitive advantage. The key aspects to focus on are:


The error message you're encountering in your Jest test for a React component indicates an issue related to the `<Router>` component from `react-router`. This kind of error often occurs when there's a problem with the way routing is set up or used in your test environment. Let's break down the error and see how you can address it:


Certainly! Let's consider a simple example using a fictional coffee shop to understand how Customer Lifetime Value (CLV) can be calculated and its relevance to market segmentation:
The Coffee Shop Example
Scenario:
Let's say "Bean There" is a small coffee shop.
They have identified two main customer segments:
1. Regulars: Customers who come in daily for their coffee fix.
2. Occasionals: Customers who visit occasionally, perhaps on weekends or special occasions.
Data Collection:
Regulars:
Average spend per visit: $5
Visits per week: 7 (daily)
Estimated customer relationship duration: 4 years
Occasionals:
Average spend per visit: $15 (they often buy pastries or bring friends)
Visits per month: 2
Estimated customer relationship duration: 3 years