Text classification with LLMs

Jean-Baptiste GIDROL[a]

[a]*Data Science Pole, OpenClassroom in partnership with CentraleSupélec,*

Abstract

In the evolving landscape of natural language processing, the application of large language models (LLMs) such as GPT-3.5 and GPT-4 offers a fresh perspective on text classification tasks. In my research, I looked into using LLMs to categorize product names into seven predefined categories. Through meticulous instruction and consideration in answer formatting, I attempted to harness the capabilities of GPT-4. For a more rounded understanding, I compared my results with well-established text transformation techniques, including TfidfVectorizer, Word2vec, BERT, and USE. My observations indicate that while LLMs are promising, the essence of obtaining valuable outcomes is in presenting them with the right context and sculpting their responses effectively. This comparison not only highlights the potential of LLMs but also underscores the nuances of engaging meaningfully with these advanced models. Through this paper, I hope to offer insights and reflections to the broader community, emphasizing the practicality and intricacies of deploying LLMs. All associated notebooks and materials for this research can be accessed at the following repository: `https://github.com/Noxfr69/LLM_text_classification`.

Keywords:
Large Language Models (LLMs), Text Classification, Contextual Interaction, Traditional Text Transformation Techniques

1. Introduction

Text classification, an integral sub-field of Natural Language Processing (NLP), has traditionally employed a plethora of techniques to categorize text into defined groups or classes. These techniques, ranging from simpler word counting methods to more sophisticated embeddings, have proven their utility over time. However, with the advent and subsequent iterations of Large Language Models (LLMs) like GPT-3.5 and GPT-4, there arises a potential paradigm shift in how we approach text classification tasks.

Given the innate capability of LLMs to understand context and produce human-like textual responses, this research explores the efficacy of using such models, specifically GPT-4, to classify product names into predefined categories. More than a mere experiment, this paper aims to understand the nuances of engaging with LLMs, emphasizing the importance of providing them the right context and appropriately formatting their outputs.

While traditional techniques remain valuable for a range of applications, understanding the comparative strengths and weaknesses of LLMs in the realm of text classification can offer significant insights. By juxtaposing LLMs with established text transformation techniques, this paper aims to illustrate the approach to text classification using GPT, offering insights and practical methods that readers can experiment with in their own endeavors.

2. The dataset

Our dataset consists of 1,050 items listed on an online shop, each accompanied by its respective category, product name, and product description. It's important to note that this dataset, like many real-world datasets, is not without its imperfections. There may be instances where items are categorized inaccurately. However, the primary objective here is not to achieve perfect classification but to use this dataset as a consistent baseline. By doing so, we aim to compare the performance of various algorithms and discern if one method significantly outshines the others in terms of classification accuracy.

2.1. Data Limitations and Token Constraints

In our analysis, we utilize both the product name and product description when employing conventional algorithms. However, when turning our attention to large language models, especially those accessed through paid APIs, we encounter practical constraints. Specifically, these models come with token limitations, which directly influence the cost associated with their usage. Given these limitations, and to maintain feasibility in terms of both computation and expenses, we've decided to exclusively use the product name for our experiments with the GPT model.

3. Chat approach

In the vast expanse of literature surrounding natural language processing, the mechanics, architecture, and intricacies of large language models (LLMs) like GPT variants have been discussed extensively. Furthermore, the functional aspects of 'ChatGPT' and its implementations are now well-established within the academic and developer communities. Given this backdrop, I've decided to refrain from revisiting these foundational topics, as they've been comprehensively covered

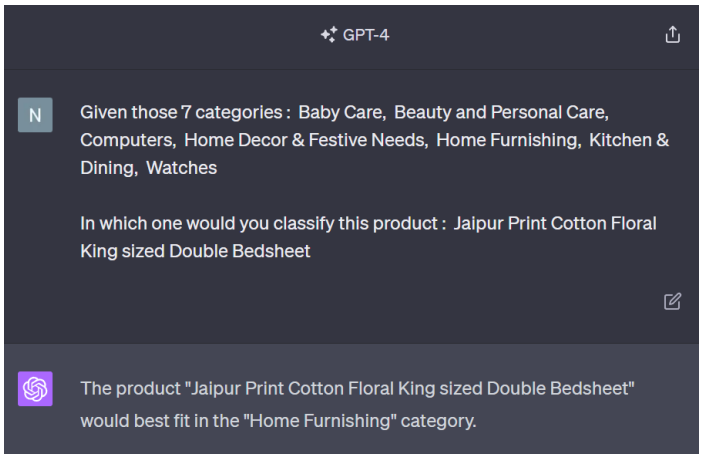

Figure 1: Asking chatgpt for classification

in previous works. Instead, this paper will pivot directly to the heart of the matter at hand: setting up a classification problem using an LLM, delving into the nuances, strategies, and insights that can assist others in similar endeavors.

In the preceding figure 1, we illustrate the process of requesting ChatGPT to classify a product among seven distinct categories. By now, the capability of this model to perform such tasks is well-acknowledged, and it's intuitively understood that leveraging ChatGPT offers a viable method for text classification. However, while the interactive chat format is insightful, it's impractical for large-scale applications. Manually inputting each product individually is not only tedious but also time-consuming. Furthermore, one of the significant challenges with ChatGPT and similar LLMs is the consistency of their output, making the extraction and utilization of results particularly challenging

4. Setting up a Classification Task with LLMs

Before diving deep into the application of Large Language Models (LLMs) like GPT for text classification, it is pivotal to ensure the foundational tools and configurations are appropriately set up. Our primary channel to communicate with the GPT models is through OpenAI's API, a powerful interface tailored to interact seamlessly with these mammoth models.

```
import openai
import json
import pandas as pd
openai.api_key = OPENAI_API_KEY
```

For those new to this, it's imperative to ascertain that all necessary Python libraries are present in your environment. If you find any missing, a simple pip installation should suffice:

```
pip install openai pandas
```

However, a word of caution is warranted here. The placeholder OPENAI-API-KEY needs to be replaced with your unique API key associated with your OpenAI account. While many might assume that a subscription to GPT-4 covers all costs, the reality is a tad different. OpenAI's API usage, especially for extensive tasks, incurs additional charges. It's always a prudent strategy to stay updated with OpenAI's latest pricing policies to prevent any unexpected expenses.

Certainly! I'll provide a more detailed and comprehensive breakdown of the ChatCompletion.create() function and its intricacies.

4.1. Designing the Query for GPT

Engaging with GPT, particularly the chat-orientated versions, demands a meticulous and informed approach to crafting the right queries. As illuminated by Törnberg (2) in his detailed exploration on utilizing LLMs for text analysis, the nuances of building a potent `ChatCompletion.create()` query are paramount. This function, a centerpiece of the OpenAI API, facilitates a dynamic conversation with GPT, mirroring a chat interface:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": "You
             are a helpful assistant."},
        {"role": "user", "content": "Hello
            , assistant."},
        {"role": "assistant", "content": "
            Hello! How can I assist you
            today?"},
        {"role": "user", "content": "
            Classify this product: XYZ"}
    ]
)
```

The Conversational Structure:

- *Roles*: We interact with three key roles in our setup: 'system', 'user', and 'assistant'. The 'system' provides high-level directives for the conversation, 'user' sends queries, and the 'assistant' (GPT) furnishes the responses.
- *Conversation Flow*: This method encourages a dialogic style. Previous messages influence GPT's context and response, enabling the user to reference earlier messages or elucidate statements, akin to standard chat interactions.
- *Message Composition*: Each message is a dictionary with a role and content. The 'role' indicates the sender, while 'content' provides the textual message.

Crafting Effective Queries:

- *Initiating Conversations*: Starting with a directive, such as a system message, can steer the conversation. Simple greetings can also enhance response quality.

- *Contextual Understanding*: The conversation's history is crucial. GPT retains earlier context, enabling coherent responses to user references.

- *Guidance and Specificity*: Clear and direct user requests yield more precise responses. Ambiguities might lead to broad answers.

Model Selection: The 'model' parameter lets users specify the GPT version. Here, "gpt-3.5-turbo" is chosen. In this paper we will look to compare the result of "gpt-3.5-turbo" and "gpt4"

In summation, the `ChatCompletion.create()` function offers a dynamic medium to exploit GPT's capacities. Proper chat structuring significantly influences the response's relevance and quality.

4.2. Handling Varied Outputs from ChatGPT and Other LLMs

The flexibility inherent to LLMs can be both a boon and a challenge. Their ability to generate diverse outputs suits many applications, but when it comes to tasks that require consistent and specific outcomes, such as classification, we must navigate them with precision.

Simon Willison, a notable figure in the realm of programming and web technologies, remarked, "LLMs are very good at returning format like JSON, which is really useful for writing code that uses them" (1). Building upon this insight, one effective strategy when working with GPT and similar models is to guide the output towards a well-defined structure, like JSON. Instructing GPT to produce a structured response, such as `"category": "electronics"`, not only guarantees consistency but also simplifies subsequent data handling processes.

An additional layer of control over the model's output is provided by the temperature setting. A higher value introduces more variability, harnessing GPT's creative potential, while a lower value ensures a more predictable and deterministic response.

Furthermore, the strategy of guiding outputs towards well-defined structures like JSON can synergize well with the concept of batch prompting, as explored in the paper "Batch-Prompt: Accomplish more with less" by Lin et al. (3). In our specific application, each JSON line or key-value pair could be conceived as a singular prompt within a larger batch. By structuring prompts in this manner, we not only enhance the computational efficiency but also enforce a level of consistency across the outputs.

5. Understanding ChatCompletion Output

Having now grasped the essentials of making an API call, our focus turns to the effective utilization of the response or output derived from the API. Upon initiating the API call, GPT responds with a data structure that encapsulates its response, among other details.

To visualize the raw output from the model, one might employ:

```
print(response)
```

The resultant output has the following structure:

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "this XYZ product is in
             the category X",
        "role": "assistant"
      }
    }
  ],
  "created": 1692435805,
  "id": "chatcmpl-7
     pC9dUgRf0X7CnsnLHYFJPJHiQFWD",
  "model": "gpt-3.5-turbo-0613",
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 37,
    "prompt_tokens": 195,
    "total_tokens": 232
  }
}
```

A brief rundown of the key components:

- **choices**: Contains the primary output message from the assistant. Nested within, the 'content' key provides the model's response.

- **created**: A timestamp indicating the time of response creation.

- **id**: A unique identifier for the API call.

- **model**: Indicates the specific GPT model version used for the response.

- **usage**: Details about token consumption, broken down into completion tokens, prompt tokens, and the total.

For most applications, the primary focus is on the 'content' within the 'choices' segment. However, understanding the complete structure is invaluable for comprehensive response handling and potential debugging.

5.1. Accessing the Message Content

In most interactions with GPT via the OpenAI API, the model typically returns a single response or "choice". This streamlined response pattern simplifies the extraction process for the primary content of interest.

To extract the content or answer string from the response:

```
content = response.choices[0].message.
    content
print(content)
```

Executing the above will display the model's generated response. In our example, the output will be:

> "this XYZ product is in the category X"

This enables straightforward further processing or direct utilization. Note that this assumes a singular choice in the model's reply. If multiple choices are specified when making the request, it's prudent to iterate over each choice and process them individually.

6. Building the Text Classification Logic

Having grasped the intricacies of the OpenAI API, it's clear that our input needs to be string-based, given that we can't directly send dataframes. Yet, by manipulating our data appropriately, we can make this process smooth, efficient, and quite enjoyable!

6.1. Building the Input

Our primary task is to convert the data from the dataframe into a format GPT can understand and process. We want to extract product names and associate them with a unique index, creating a serialized string of products:

```
# Convert the product_name column into a
    numbered list format
formatted_product_names = [f"{i}: {x}" for
     i, x in enumerate(text_data['
    product_name'].head(5))]

# Convert the entire series into a single
    string
products_string = ', '.join(
    formatted_product_names)
```

The resultant string appears as:

> '0: Elegance Polyester Multicolor Abstract Eyelet Door Curtain, 1: Sathiyas Cotton Bath Towel, 2: Eurospa Cotton Terry Face Towel Set, 3: SANTOSH ROYAL FASHION Cotton Printed King sized Double Bedsheet, 4: Jaipur Print Cotton Floral King sized Double Bedsheet'

6.2. Setting the Context Right

The essence of unlocking GPT's full potential lies predominantly in the art of crafting the right dialogue or, more aptly, the 'prompt'. Ensuring that our instructions to GPT are both clear and precise is crucial, as the model's performance hinges on the specificity and clarity of these directives. By introducing a system role at the outset, we grant the assistant an overarching orientation, a compass of sorts, guiding its responses in the desired direction. It's not just about telling the model what to do, but more importantly, setting the stage for how to think about the task at hand. Here's the magic unravelled:

```
messages=[
    {"role": "system", "content": "You are
         a helpful assistant tasked to
        classify text into 7 categories."
        },
    {"role": "user", "content": "Here is a
         list of the categories: 1: Baby
        Care, 2: Beauty and Personal Care,
         3: Computers, 4: Home Decor &
        Festive Needs, 5: Home Furnishing,
         6: Kitchen & Dining, 7: Watches"
        },
    {"role": "assistant", "content": "How
        should I format my answer?"},
    {"role": "user", "content": "Answer in
         a JSON formatting style {\"
        sentence index\" : category number
        }"},
    {"role": "assistant", "content": "Okay
        , give me sentences to categorize"
        },
    {"role": "user", "content": f"{
        products_string}"}
]
```

Remember, it's not just about posing a question; it's about **sculpting the context** in which GPT operates, ensuring it aligns perfectly with our objectives. **Prompting** is what makes or breaks the use of these incredible models. They are not merely 'prompting tricks'; it's really the essence of LLMs. You can even liken it to **hyperparameters tuning**.

6.3. Handling JSON Output

The beauty of our interaction is that we've guided the model to return a JSON formatted output. Parsing this becomes a breeze:

```
result = {}
result = json.loads(response.choices[0].
    message.content)
print(result)
```

The results are pretty cool if I can say so myself:

{'0': 4, '1': 1, '2': 1, '3': 4, '4': 4}

Can you grasp the elegance? We've just harnessed the raw power of GPT, parsed our text, and got classified results in an instant. The possibilities are both fascinating and endless!

7. Token Limitations

The token limitation is a critical aspect to consider when working with Large Language Models (LLMs) like GPT-3.5 and GPT-4. Specifically, the GPT-3.5 model has a context window of 4K tokens, while the base version of GPT-4 offers an 8K token limit. This means that both the context and the input strings cannot exceed these token limits.

Given the vast size of datasets commonly used in real-world scenarios, it becomes crucial to handle this limitation effectively. One approach is to partition the dataset and process it in manageable chunks, ensuring that the token limits are not breached.

```
i = 0
while i < text_data.shape[0]:
    # Take the text_data in chunks of 100
        rows to stay within the token
        limit
    end_index = min(i + 100, text_data.
        shape[0]) # Handle the scenario
        with less than 100 rows left

    # Select the current chunk of data
    current_chunk = text_data['
        product_name'].iloc[i:end_index]

    formatted_product_names = [f"{idx+i}:
        {x}" for idx, x in enumerate(
        current_chunk)]
    products_string = ', '.join(
        formatted_product_names)

    # Here, we would make an API call with
         the 'products_string'
    # response = openai.ChatCompletion.
        create(.....)

    # Progress to the next chunk of data
    i += 100
```

This partitioning method ensures that our application remains efficient and doesn't exceed the model's token constraints. For a comprehensive look into this approach, including other nuances and optimizations, please refer to the detailed notebook available in the GitHub repository (4).

8. Exploiting the JSON Output

Once we've obtained our results using the methodology described earlier, the next step is to process these results to derive meaningful insights. This involves loading the obtained JSON outputs, converting them into structured formats suitable for comparison, and then conducting the actual comparison.

8.1. Loading and Structuring the Data

The results from both GPT-3.5 and GPT-4 can be fetched directly from the associated repository (4). For the purpose of this section, we'll be considering the GPT-4 results as an example:

```
# Loading the GPT-4 results
with open('./result_gpt4.json', 'r') as f:
    result_gpt4 = json.load(f)

# Converting the results into a DataFrame
    for easier manipulation
df_classification_gpt4 = pd.DataFrame(list
    (result_gpt4.items()), columns=['
    Product Index', 'Category Number'])
```

8.2. Comparing with Ground Truth using ARI

With our predictions neatly structured in a DataFrame, it's time to compare them against the actual categories using metrics. One of the popular metrics for this task is the Adjusted Rand Index (ARI). ARI measures the similarity between two data clusterings, adjusted for chance. An ARI score of 1 indicates perfect agreement, while a score of 0 suggests a random assignment.

```
# Calculating the Adjusted Rand Index
rand_index = adjusted_rand_score(text_data
    ['product_category_tree'],
    df_classification_gpt4['Category
    Number'])
print(f'Rand index for gpt4: {rand_index}'
    )
```

By obtaining the ARI score, we get a quantitative measure of how well the model's predictions align with the actual categories. This allows for objective evaluations and facilitates potential iterative improvements in the future.

9. Comparing results

Before we delve deep into our LLMs' results, it's essential to revisit the foundational models that have been at the forefront of text classification for years.

9.1. *TfidfVectorizer*

TfidfVectorizer is based on the principle that words which appear frequently in a document, but not too often in many documents, are significant. It quantifies the importance of a term in a document relative to a whole corpus, offering a numerical representation that can be fed into classical machine learning algorithms.

9.2. *Word2Vec*

Word2Vec is a step up in representing text. Instead of counting word occurrences, it captures the semantics of words by placing them in a high-dimensional space. In this space, words with similar meanings are closer to each other. It's a leap from counting words to understanding them in context.

9.3. *BERT*

BERT reshaped the NLP scene by using bidirectional contexts to understand words in any given text. Pre-trained on vast textual datasets, it's a model that comprehends the subtle nuances of language. The model can then be fine-tuned for specific tasks, showcasing its versatility.

9.4. *USE (Universal Sentence Encoder)*

The Universal Sentence Encoder goes beyond word representations. It captures the essence of entire sentences, converting them into dense vectors. This model is particularly effective when one needs a consistent and robust representation of text without diving deep into individual word semantics.

9.5. *Results grand reveal*

With this foundational understanding, let's see how these tried-and-true models measure up against our latest entrants, GPT-3.5 and GPT-4. The competition is indeed intriguing!

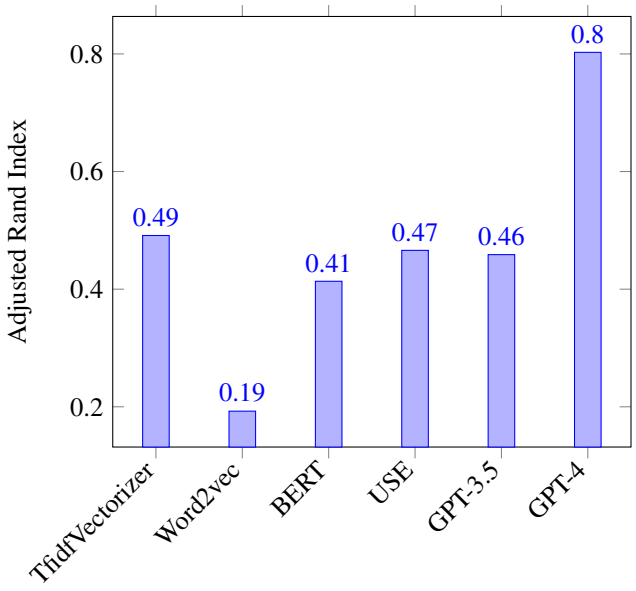

Figure 2: Comparison of text classification models using ARI

The graph starkly illustrates the remarkable progress between GPT-3.5 and GPT-4. GPT-4's performance surpasses not only its predecessor but also most traditional models, showcasing its prowess in text classification. It's noteworthy to mention that GPT-4's results hint at an interesting possibility: we might be at a juncture where the model's predictions could be considered more reliable than certain categorizations in imperfect datasets. GPT-4's advanced capabilities suggest it might discern nuances or correct errors that might elude many. This underscores the transformative potential of such models in refining and improving the quality of text-related tasks.

10. Conclusion

The world of Natural Language Processing has seen revolutionary advancements in recent times, notably with the introduction of Large Language Models (LLMs) such as GPT-3.5 and GPT-4. Our exploration into these models, especially in the domain of text classification, offered insightful takeaways.

GPT-4, in particular, showcased a substantial leap in classification capabilities. Its performance, as quantified by the Adjusted Rand Index, not only surpassed its predecessor, GPT-3.5, but also eclipsed several traditional models. This progression in a single generational leap from GPT-3.5 to GPT-4 suggests a promising trajectory for future iterations of the GPT series.

Traditional models, including TfidfVectorizer, Word2Vec, BERT, and the Universal Sentence Encoder, have been foundational in our understanding of text data. While they remain relevant, the prowess of GPT-4 sets a new benchmark. Its adaptability, combined with the power of prompt design, paints an optimistic picture of its applicability across various textual domains.

As the capabilities of LLMs continue to grow, there is increasing hope that future models might evolve into comprehensive solutions for diverse textual challenges. With the trend observed from GPT-3.5 to GPT-4, it's not far-fetched to envision subsequent models providing unparalleled text-related solutions.

In summary, our exploration underscores the vast potential of modern LLMs. As we anticipate further advancements, the reliance on these models for holistic text-related tasks seems not only feasible but inevitable.

Acknowledgements

I would like to thank my mentor during my master's degree, Benjamin Tardy, for his guidance and support. A special mention goes to Shuai Wang from the Technical University of Munich for introducing me to the world of data science last year. Additionally, appreciation is extended to Simon Willison for his enlightening articles and videos on LLMs.

References

[1] Willison, S. (2023). The weird world of LLMs: Tips for using them. Retrieved from https://simonwillison.net/2023/Aug/3/weird-world-of-llms/#tips-for-using-them

[2] Törnberg, P. (2023). How to use LLMs for Text Analysis. *arXiv preprint arXiv:2307.13106*. Retrieved from https://arxiv.org/abs/2307.13106

[3] Jianzhe Lin, Maurice Diesendruck, Liang Du, Robin Abraham. (2023). BatchPrompt: Accomplish more with less. *arXiv preprint arXiv:2307.13106*. Retrieved from https://arxiv.org/abs/2309.00384

[4] *Text Classification with LLMs*. [githubrepo]. Available: https://github.com/Noxfr69/LLM_text_classification