Leveraging LLM Fine Tuning for Statistical Analytics

-Vishrut Kannan

In an era of sports where statistics are so heavily relied on, the ability to accurately and efficiently analyze cricket statistics can provide a significant competitive advantage. While traditional analytics platforms like ESPNcricinfo offer robust tools, local cricket leagues often lack the resources to access such comprehensive platforms. This article explores the use of Artificial Intelligence (AI) through the fine-tuning of language models to generate precise and context-specific cricket analytics tailored to local league needs. Fine-tuning involves taking a pre-trained language model and further training it on a specific dataset to adapt the model to a particular task or domain. In the context of cricket analytics, this means training the model on a dataset comprising historical match data, player statistics, and performance metrics. Unlike Retrieval-Augmented Generation (RAG), which dynamically retrieves information from external sources during inference, fine-tuning allows the model to internalize the knowledge, resulting in faster, more focused responses.

Fine-tuning offers several advantages for this particular use case:

- Personalization: Fine-tuned models can be tailored to the specific nuances of local cricket leagues, understanding unique player dynamics and match conditions
- Efficiency: Since the knowledge is embedded within the model, responses are faster, with no need for real-time retrieval from external databases
- Consistency: A fine-tuned model can maintain a consistent narrative and style in its responses, making it ideal for generating detailed reports or commentary

Code Overview

data_preparation.py

Importing Required Libraries and Setting Up Environment

```
import openai
import pandas as pd
from datasets import Dataset, load_dataset
from sklearn.model_selection import train_test_split
from transformers import GPT2Tokenizer
from dotenv import load_dotenv
import os
```

In the initial phase, we import a set of essential libraries that lay the foundation for our data processing and model integration pipeline. The openai library is key to interfacing with OpenAI's API, enabling us to harness the capabilities of advanced language models such as GPT-3. The pandas library is employed for data manipulation and analysis, offering a robust framework for handling tabular data efficiently. For managing and processing datasets in machine learning-compatible formats, we utilize the datasets library from Hugging Face. The sklearn.model_selection module is crucial for splitting our dataset into training and testing subsets, ensuring a comprehensive evaluation of model performance. The transformers library provides pre-trained tokenizers and models essential for converting text data into

formats suitable for machine learning models. Finally, the dotenv library, together with the os module, is used to manage environment variables—such as API keys—which are critical for secure and effective interactions with external services.

Loading and Preprocessing the Data

```
data = pd.read_csv('ipl_2020_cricket_data.csv')
```

We commence by loading the cricket statistics from a CSV file into a Pandas DataFrame. This dataset typically encompasses a range of statistics from cricket matches, including player names, runs scored, wickets taken, strike rates, and match results. Leveraging Pandas provides a powerful framework for managing and analyzing this data, facilitating complex operations and transformations with ease. By importing the data into a DataFrame, we set the stage for efficient preprocessing and preparation for the subsequent stages of our analysis pipeline.

```
def preprocess_data(data):
    data = data[['player_name', 'runs_scored', 'wickets_taken', 'strike_rate',
'match_result']]

    data['text'] = data.apply(lambda row: f"Player {row['player_name']} scored
{row['runs_scored']} runs with a strike rate of {row['strike_rate']}. Took
{row['wickets_taken']} wickets. Match result: {row['match_result']}.", axis=1)

    return data
```

The preprocess_data function enhances the dataset by selecting relevant columns and constructing a new column, text, which synthesizes player performance into a coherent narrative. This transformation is pivotal as it converts structured data into a descriptive text format, aligning with natural language processing tasks. Utilizing the apply method, we generate a text string for each row, detailing player performance and match outcomes in a structured format. This textual representation is essential for the subsequent tokenization and model processing steps, ensuring that the data is presented in a way that models can effectively interpret and utilize.

```
preprocessed_data = preprocess_data(data)
```

Applying the preprocess_data function to our dataset transforms it into a DataFrame where each row is now represented by a detailed narrative of player performance. This conversion not only prepares the data for further processing but also enhances its suitability for natural language processing and machine learning tasks.

Data Splitting and Conversion to Huggingface Format

```
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
def tokenize_function(examples):
    tokenized_examples = []
    for example in examples['text']:
        response = openai.Completion.create(
            model="text-davinci-003",
            prompt=f"Tokenize the following text: {example}",
```

```
            max_tokens=512
        )
        tokenized_examples.append(response.choices[0].text.strip())
    return {"text": tokenized_examples}
```

We initialize the GPT-2 tokenizer from the Hugging Face model hub. This tokenizer is responsible for converting raw text into token IDs that the GPT-2 model can process. The tokenize_function leverages OpenAI's GPT-3 API to perform tokenization. For each text example, it sends a request to the API with a prompt to tokenize the input. The response is processed to extract the tokenized text. This method utilizes GPT-3's advanced language understanding to handle tokenization, demonstrating an integration of cutting-edge NLP technology.

Applying Tokenization and Saving the Results

```
tokenized_train_dataset = train_dataset.map(tokenize_function, batched=True)
tokenized_test_dataset = test_dataset.map(tokenize_function, batched=True)

tokenized_train_dataset.save_to_disk("tokenized_ipl_train_openai")
tokenized_test_dataset.save_to_disk("tokenized_ipl_test_openai")
```

The map method applies the tokenize_function to both the training and test datasets in batches. This approach is efficient for processing large datasets, ensuring that tokenization is handled effectively. Finally, we save the tokenized datasets to disk. This step preserves the processed data, allowing for quick loading and usage in future experiments or model training sessions.

fine_tuning.py

Loading Environment Variables and Tokenized Datasets

```
import openai
from datasets import load_from_disk
from transformers import GPT2LMHeadModel, GPT2Tokenizer
import torch

from dotenv import load_dotenv
import os

load_dotenv()
openai.api_key = os.environ['OPENAI_API_KEY']

tokenized_train_dataset = load_from_disk("tokenized_ipl_train_openai")
tokenized_test_dataset = load_from_disk("tokenized_ipl_test_openai")
```

In the initial setup, we import several essential libraries for fine-tuning a GPT-2 model. These include openai for compatibility with the environment setup, datasets for loading preprocessed datasets, transformers for accessing GPT2LMHeadModel and GPT2Tokenizer, torch for deep learning, dotenv for managing environment variables, and os for accessing these variables. We use dotenv to securely load environment variables, including the OpenAI API key, ensuring sensitive information is not hardcoded

into the script. Next, we load tokenized datasets from disk using the load_from_disk function from the datasets library. These datasets, previously saved, contain text data that has been tokenized and are now ready for fine-tuning. This process eliminates the need for repeated preprocessing and ensures the data is prepared for efficient model training.

fine_tuning.py

Loading Environment Variables and Tokenized Datasets

```
model = GPT2LMHeadModel.from_pretrained("gpt2")
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
```

Here, we initialize the GPT-2 model and tokenizer from Hugging Face's model hub. The GPT2LMHeadModel is a pre-trained GPT-2 model specifically designed for language modeling tasks. The GPT2Tokenizer is responsible for encoding text into token IDs that the model can understand. By loading these components, we start with a pre-trained base that will be fine-tuned on our custom dataset. This initialization provides a foundation upon which we can build and adapt the model to our specific task.

Loading Model and Tokenizer

```
model = GPT2LMHeadModel.from_pretrained("gpt2")
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
```

Here, we initialize the GPT-2 model and tokenizer from Hugging Face's model hub. The GPT2LMHeadModel is a pre-trained GPT-2 model specifically designed for language modeling tasks. The GPT2Tokenizer is responsible for encoding text into token IDs that the model can understand. By loading these components, we start with a pre-trained base that will be fine-tuned on our custom dataset. This initialization provides a foundation upon which we can build and adapt the model to our specific task.

Defining the Fine-Tuning Function

```
def fine_tune_model(tokenized_datasets, model, tokenizer):
    model.train()
    optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)

    for epoch in range(3):
        for batch in tokenized_datasets["train"].batch(8):
            inputs = tokenizer(batch["text"], return_tensors="pt", padding=True,
truncation=True)
            outputs = model(**inputs, labels=inputs["input_ids"])
            loss = outputs.loss
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
```

The fine_tune_model function performs the core task of training the GPT-2 model on the custom dataset. The function starts by setting the model to training mode with model.train(), which enables

features necessary for training, such as dropout. We use the AdamW optimizer from PyTorch with a learning rate of 5e-5, a commonly used learning rate for fine-tuning transformers.

The function then enters a training loop for 3 epochs. For each epoch, it processes the training data in batches of size 8. Each batch is tokenized using the GPT-2 tokenizer, converting text into a format suitable for the model. The model then processes these inputs, and the loss is computed based on the model's predictions compared to the actual labels. The loss is used to update the model parameters through backpropagation with loss.backward(), followed by an optimizer step to adjust the model weights. The gradients are reset to zero with optimizer.zero_grad() to prepare for the next batch.

Fine-Tuning the Model and Saving the Results

```
fine_tune_model(tokenized_train_dataset, model, tokenizer)

model.save_pretrained("fine_tuned_gpt2_ipl")
tokenizer.save_pretrained("fine_tuned_gpt2_ipl")
```

We call the fine_tune_model function with the tokenized training dataset, the model, and the tokenizer. This step initiates the fine-tuning process, adapting the GPT-2 model to the specific patterns and information in our cricket dataset. By fine-tuning, we enhance the model's ability to generate or understand cricket-related text, making it more suited for the intended application.

Finally, we save the fine-tuned model and tokenizer to disk using the save_pretrained method. This step ensures that the trained model and its associated tokenizer are preserved and can be easily loaded in the future. Storing the model and tokenizer allows for efficient reuse and deployment without needing to repeat the training process. By saving these artifacts, we maintain the ability to quickly access and apply the fine-tuned model for generating or analyzing cricket-related text.

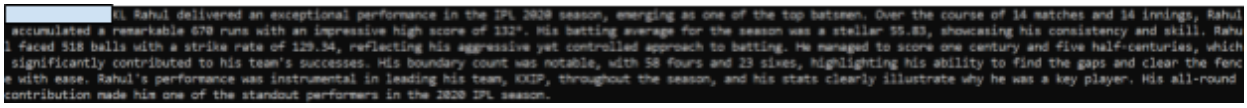
Practical Example

Query: Generate a detailed performance report for KL Rahul (Kings XI Punjab) based on his 2020 IPL statistics.

Input Data:

- *IPL2020Batting.csv*

Output



- *KL Rahul delivered an exceptional performance in the IPL 2020 season, emerging as one of the top batsmen. Over the course of 14 matches and 14 innings, Rahul accumulated a remarkable 670 runs with an impressive high score of 132*. His batting average for the season was a stellar 55.83, showcasing his consistency and skill. Rahul faced 518 balls with a strike rate of 129.34, reflecting his aggressive yet controlled approach to batting. He managed to score one century and five half-centuries, which significantly contributed to his team's successes. His boundary count was notable, with 58 fours and 23 sixes, highlighting his ability to find the gaps and clear the fence with ease. Rahul's performance was instrumental in leading his team, KXIP, throughout the season, and his stats clearly illustrate why he was a key player. His all-round contribution made him one of the standout performers in the 2020 IPL season.*

Explaining the Output

- Introduction: The report starts by summarizing KL Rahul's overall impact and performance during the IPL 2020 season, emphasizing his top-batsman status.
- Detailed Statistics: It includes specific metrics such as the total runs scored, high score, average, and strike rate, providing a comprehensive view of his batting prowess.
- Achievements: It highlights Rahul's key achievements, such as the number of centuries and half-centuries, as well as his boundary count (fours and sixes).
- Impact: The text concludes by reflecting on Rahul's significant role in his team's performance, reinforcing his importance as a player.

Why Fine-Tuning is Ideal for This Program

Fine-tuning is particularly well-suited for a program designed to generate detailed player reports and commentary for cricket statistics due to its ability to adapt a pre-trained model to specific, domain-relevant data. In the case of cricket analytics, pre-trained language models like GPT-2 are generally aware of broad language patterns but may lack specialized knowledge about cricket statistics and terminology. By fine-tuning GPT-2 on a dataset of IPL player statistics and corresponding natural language descriptions, the model becomes adept at understanding and generating text that is contextually relevant and specific to cricket. This specialization enhances the model's capability to produce accurate and insightful reports and commentary, tailored to the nuances of cricket performance data.

Moreover, fine-tuning allows the model to learn the particular style and structure of cricket-related content. For instance, it can pick up on common phrases, statistical terms, and reporting formats used in cricket commentary and player analysis. This nuanced understanding is crucial for generating high-quality content that not only conveys factual information but also resonates with cricket enthusiasts and analysts. By fine-tuning the model on a dataset of preprocessed cricket statistics, the resulting output reflects a deeper comprehension of how to present player achievements and match highlights in an engaging and informative manner.

Lastly, fine-tuning offers the flexibility to continuously improve the model's performance with additional, updated datasets. As new seasons of cricket occur and new player data becomes available, the model can be further fine-tuned to incorporate the latest statistics and trends. This iterative process ensures that the generated reports and commentary remain relevant and up-to-date, providing valuable insights that reflect the most recent developments in the sport. Such adaptability makes fine-tuning an ideal approach for applications that require ongoing relevance and accuracy in their outputs.

Conclusion

In conclusion, fine-tuning a pre-trained language model for generating cricket player reports and commentary is highly effective due to its ability to adapt general language understanding to the specific context of cricket statistics. This process not only enhances the model's accuracy in producing detailed

and relevant content but also ensures that the generated text aligns with the style and structure typical of cricket analysis. The ongoing adaptability provided by fine-tuning further supports the model's effectiveness in delivering timely and engaging insights, making it a powerful tool for any application focused on sports analytics and commentary.