**User**

Hey there, I’m trying to write Python code to load a BERT model and then apply LIME (more specifically, its textual explainer) to it. This is what I have so far:

**Bing**

Sure, I’d be

**User**

Hey there, I’m trying to write Python code to load a BERT model trained on financial data, called finBERT, and then apply LIME (more specifically, its textual explainer) to it. This is what I have so far:

# Import PyTorch

import torch

# Import Huggingface Transformers library for loading the model

from transformers import AutoTokenizer, AutoModelForSequenceClassification

# select tokenizer

tokenizer = AutoTokenizer.from_pretrained(“ProsusAI/finbert”)

# select model

model = AutoModelForSequenceClassification.from_pretrained(“ProsusAI/finbert”)

# Input the text sequence we want finBERT to classify into the tokenizer

input = tokenizer(“Nvidia’s market value surpassed those of fellow technology titans Amazon and Alphabet on Monday, an almost unbelievable feat accomplished as Nvidia’s stock more than quadrupled over the last 15 months as investors bought into Nvidia’s market-leading position in artificial intelligence.”, return_tensors=“pt”)

First off, can you please explain to me what return_tensors=“pt” is about? Following that, could you please help me continue my code so that finBERT classifies my input and then a LIME textual explainer gets applied?

**Bing**

Sure, I’d be happy to help!

The `return_tensors="pt"` argument in the `tokenizer` function is used to specify the desired format for the returned tensors. Here, `"pt"` stands for PyTorch, which means the output will be PyTorch tensors. This is necessary because the BERT model (in this case, finBERT) from Hugging Face’s transformers library expects input in this format.

Now, let’s continue with your code. After tokenizing your input, you can feed it into your model for classification. Then, you can initialize a LIME TextExplainer and use it to explain the model’s predictions. Here’s how you can do it:

```python
# Import lime
from lime.lime_text import LimeTextExplainer

# Get the model's output (logits)
output = model(**input)

# Get the probabilities by applying the softmax function to the logits
probs = torch.nn.functional.softmax(output.logits, dim=-1)

# Get the predicted class
predicted_class = torch.argmax(probs)

# Initialize LIME text explainer
explainer = LimeTextExplainer(class_names=["Negative", "Neutral", "Positive"])

# Define a prediction function
def predictor(texts):
    inputs = tokenizer(texts, return_tensors="pt", padding=True, truncation=True)
    outputs = model(**inputs)
    probs = torch.nn.functional.softmax(outputs.logits, dim=-1)
    return probs.detach().numpy()

# Explain the model's prediction
explanation = explainer.explain_instance("Nvidia’s market value surpassed those of fellow technology titans Amazon and Alphabet on Monday, an almost unbelievable
                                         predictor,
                                         num_features=6,
                                         labels=(predicted_class.item(),))

# Print the explanation
print(explanation)
```

This code will give you an explanation of the model’s prediction for your input text. The `num_features` parameter in the `explain_instance` method determines the number of features (i.e., words or phrases) that contribute most to the model’s prediction. You can adjust this number as needed.

Please note that you might need to install the `lime` package if you haven’t done so already. You can do this by running `pip install lime`. Also, this is a basic example and might need to be adjusted based on your specific use case and data.

I hope this helps! Let me know if you have any other questions. 😊

**User**

Awesome, thank you, this is super helpful! For context, with this code I just want to try out LIME and see what explainable AI looks like in action. I am a beginner to Python programming. Could you please explain to me what the PyTorch tensors do in this code?

**Bing**

Absolutely, I'd be happy to explain!

In the context of your code, PyTorch tensors are multi-dimensional arrays with a uniform type (i.e., all elements in the tensor have the same data type). They are similar to NumPy's ndarrays, but unlike NumPy, PyTorch tensors can utilize GPUs to accelerate their numeric computations.

Here's what the PyTorch tensors are doing in your code:

1. **Tokenization**: The `tokenizer` function converts your input text into a format that the model can understand. This involves splitting the text into tokens (words, subwords, or characters), and then mapping each token to a unique integer (its ID in the model's vocabulary). The output is a PyTorch tensor where each element is the ID of a token in the input text.

```
input = tokenizer("Nvidia's market value surpassed those of fellow technology titans Amazon and Alphabet on Monday, an almost unbelievable feat accomplished as N
```

1. **Model Inference**: The `model` function takes the tensor of token IDs and feeds it through the finBERT model. The model outputs logits, which are raw, unnormalized scores for each class in the model's classification task. These logits are also stored in a PyTorch tensor.

```
output = model(**input)
```

1. **Softmax Function**: The `torch.nn.functional.softmax` function takes the tensor of logits and applies the softmax function to it. The softmax function converts the logits into probabilities, which are easier to interpret. The output is a tensor of probabilities.

```
probs = torch.nn.functional.softmax(output.logits, dim=-1)
```

1. **Argmax Function**: The `torch.argmax` function takes the tensor of probabilities and finds the index (class) with the highest probability. This is the model's predicted class for the input text.

```
predicted_class = torch.argmax(probs)
```

In summary, PyTorch tensors are used to store and manipulate the data at each step of processing the input text, from tokenization to prediction. They are a fundamental part of working with neural networks in PyTorch.

I hope this helps! Let me know if you have any other questions. 😊

**User**

Oof, I tried to run our completed code in Visual Studio Code, but unfortunately there are errors. I'm not sure what's wrong. Could you please look at the following log info and help me understand what went wrong?

PS D:\AI\XAI-Projects> & C:/Users/Jana/AppData/Local/Programs/Python/Python311/python.exe d:/AI/XAI-Projects/finbert-lime.py tokenizer_config.json: 100%|██████████| 252/252 [00:00<?, ?B/s] C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\site-packages\huggingface_hub\file_download.py:149: UserWarning: `huggingface_hub` cache-system uses symlinks by default to efficiently store duplicated files but your machine does not support them in C:\Users\Jana.cache\huggingface\hub\models–ProsusAI–finbert. Caching files will still work but in a degraded version that might require more space on your disk. This warning can be disabled by setting the `HF_HUB_DISABLE_SYMLINKS_WARNING` environment variable. For more details, see https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations. To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to see activate developer mode, see this article: https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-development warnings.warn(message) config.json: 100%|██████████ 758/758 [00:00<?, ?B/s] vocab.txt: 100%|██████████| 232k/232k [00:00<00:00, 1.30MB/s] special_tokens_map.json: 100%|██████████| 112/112 [00:00<?, ?B/s] Traceback (most recent call last): File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\site-packages\urllib3\connectionpool.py", line 467, in _make_request self._validate_conn(conn) File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\site-packages\urllib3\connectionpool.py", line 1099, in _validate_conn conn.connect() File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\site-packages\urllib3\connection.py", line 653, in connect sock_and_verified = _ssl_wrap_socket_and_match_hostname( ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\site-packages\urllib3\connection.py", line 806, in _ssl_wrap_socket_and_match_hostname ssl_sock = ssl_wrap_socket( ^^^^^^^^^^^^^^^^ File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\site-packages\urllib3\util\ssl.py", line 465, in ssl_wrap_socket ssl_sock = ssl_wrap_socket_impl(sock, context, tls_in_tls, server_hostname) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\site-packages\urllib3\util\ssl.py", line 509, in _ssl_wrap_socket_impl return ssl_context.wrap_socket(sock, server_hostname=server_hostname) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\ssl.py", line 517, in wrap_socket return self.sslsocket_class._create( ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\ssl.py", line 1104, in _create self.do_handshake() File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\ssl.py", line 1382, in do_handshake self._sslobj.do_handshake() TimeoutError: _ssl.c:989: The handshake operation timed out

The above exception was the direct cause of the following exception:

Traceback (most recent call last): File "C:\Users\Jana\AppData\Local\Programs\Python\Python311\Lib\site-packages\requests\adapters.py", line 486, in send resp = conn.urlopen(

**Bing**

The error message you're seeing is related to a timeout during a handshake operation in an SSL connection. This is likely due to a network issue rather than a problem with your code.

The warning about `huggingface_hub` cache-system using symlinks is also not a critical issue. It's just informing you that your machine does not support symlinks and that the caching of files will still work but might require more space on your disk.

Here are a few things you could try to resolve the issue:

1. **Check your internet connection**: Make sure you have a stable internet connection. The error might be due to intermittent connectivity issues.
2. **Try running the code again**: Sometimes, network-related errors are temporary and can be resolved by simply retrying the operation.
3. **Check for firewalls or proxies**: If you're behind a firewall or using a proxy, it might be blocking or interfering with the connection. You might need to configure your firewall or proxy settings to allow the connection.
4. **Update your packages**: Make sure that your Python packages (like `requests`, `urllib3`, `huggingface_hub`, and `transformers`) are up to date. You can update them using pip:

```
pip install --upgrade requests urllib3 huggingface_hub transformers
```

5. **Use a VPN**: If the issue persists, it might be due to network restrictions in your location. In this case, you could try using a VPN.

Remember to replace `pip` with the appropriate command if you're using a different package manager or a virtual environment.

I hope this helps! Let me know if you have any other questions. 😊

### User

Okay, so the error didn't happen anymore when I ran our code again, which is good, but unfortunately I think my device might not have the resources needed to run the finBERT model and/or LIME on top? When I ran the code, it almost froze my laptop completely and when I did manage to open up task manager to check the performance tab, 15.8GB of 16GB RAM were in use and the HDD was at 100% as well. My GPU went up to 71°C at one point but stayed at around 47°C then. My CPU kept going anywhere from 7% to over 50%. I waited for an hour but then had to go so I had to force-shutdown my device. Do you have any insights regarding this, please?

### Bing

It sounds like your device is running out of resources when trying to run the finBERT model and LIME. Both of these tasks are computationally intensive, especially when run on large inputs or datasets. Here are a few suggestions:

1. **Reduce the size of your input**: If you're running this on a large text or a large number of texts, try reducing the size of your input. This could mean using a smaller text for testing, or splitting a large text into smaller chunks and processing them one at a time.

2. **Use a smaller model**: BERT models are quite large and resource-intensive. If possible, you might want to consider using a smaller model. Hugging Face offers several smaller versions of BERT that might be less resource-intensive.

3. **Run on a more powerful machine**: If you have access to a more powerful machine, consider running your code there. This could be a high-performance desktop computer, or a cloud-based virtual machine. Some cloud providers offer machines with high-performance GPUs, which can significantly speed up model inference.

4. **Use model quantization or pruning**: These are techniques to reduce the size of the model without significant loss in performance. However, these techniques might require a bit more experience with machine learning and PyTorch.

5. **Close other applications**: Make sure to close any unnecessary applications running on your computer to free up resources.

Remember, working with large language models like BERT can be resource-intensive, and it's not uncommon to run into issues like this. It's all part of the learning process. Keep going! 😊

### User

I thought I could run the finBERT model since the actual model file in the Huggingface repository is under 500MB and I assumed (because of my experience coming from LLMs) that this means it would use at most double that amount of memory. Was I mistaken there? Can you by chance find any info on the resources needed to run finBERT (since there was nothing about this on the Huggingface repository model card)?

### Bing

The size of the model file on disk doesn't directly correlate with the amount of memory required to run the model. When a model is loaded into memory, it often requires more space than the on-disk size due to various factors such as the need to store intermediate computations, additional data structures used by the machine learning framework (like PyTorch), and others.

Unfortunately, I couldn't find specific system requirements for running the finBERT model in the Hugging Face repository or elsewhere. However, BERT models are known to be resource-intensive due to their large number of parameters and the complexity of their architecture.

Here are a few general tips for managing memory usage when working with large models:

1. **Batch Size**: If you're processing multiple inputs at once (a "batch"), try reducing the batch size. Smaller batches require less memory but may increase the total computation time.

2. **Model Size**: Some models have smaller versions that use fewer resources. For example, 'bert-base-uncased' is smaller and less resource-intensive than 'bert-large-uncased'. However, smaller models may also be less accurate.

3. **Optimization**: Some machine learning frameworks offer optimization options that can reduce memory usage. For example, PyTorch has a `torch.utils.checkpoint` feature that can trade compute time for memory.

4. **Hardware**: If possible, consider upgrading your hardware. More RAM and a faster GPU can significantly improve performance and capacity when working with large models.

Remember, it's always a good idea to monitor your system's resource usage when running large models to ensure you're not exceeding your hardware's capabilities. If you continue to experience issues, you might want to consider using cloud-based resources, which can offer more computational power and memory. 😊

### User

Hmm, I see… are you maybe able to find system requirements for a BERT model similar to finBERT? If it helps, on the model card for finBERT, it just says that it's a finetuned version of "the BERT language model" which I'd assume to be just standard BERT.

### Bing

The system requirements for running a BERT model can vary greatly depending on the specific task, the size of the model, and the size of the input data. However, I found some general information that might be helpful:

1. **Memory**: BERT models are known to be memory-intensive. For example, the fine-tuning examples which use BERT-Base should be able to run on a GPU that has at least 12GB of RAM. However, the actual memory usage can be much higher depending on the batch size and sequence length.

2. **CPU and GPU**: BERT models can be run on both CPUs and GPUs, but using a GPU can significantly speed up model inference. The specific requirements will depend on the size of the model and the task.

3. **Disk Space**: The on-disk size of a BERT model can be several hundred MBs (like the finBERT model you're using), but the actual disk space usage can be higher when the model is loaded into memory and when intermediate computations are stored.

4. **Software**: You'll need a machine with Python and PyTorch installed. The Hugging Face Transformers library, which provides the BERT model, requires Python 3.6 or later and PyTorch 1.1.0 or later.

Please note that these are just general guidelines and the actual requirements can vary. If you're running into resource issues, you might need to adjust the batch size, sequence length, or use a machine with more resources.

I hope this helps! Let me know if you have any other questions. 😊

### User

Is there any way I could see how much memory finBERT wants to use when I run it? Not how much it is currently using on my device but how much it would want to use if it could?

**Bing**

Estimating the exact memory usage of a model like finBERT before running it can be challenging due to various factors such as the size of the input data, the batch size, and the specific operations performed by the model. However, there are some general methods you can use to get an approximate idea:

1. **Parameter and Buffer Sizes**: You can calculate the memory requirements of all parameters and buffers in the model. Each parameter in a BERT model is a tensor, and you can find its size in memory by multiplying the number of elements by the size of each element (which is typically 4 bytes for float32). You would need to do this for all parameters and buffers in the model.

2. **Intermediate Activations**: During the forward and backward passes, the model also needs to store intermediate activations which can take up a significant amount of memory. The memory required for these can be roughly estimated as the number of activation units (neurons) multiplied by the batch size multiplied by the size of each activation (again, typically 4 bytes for float32).

3. **Profiling Tools**: There are also profiling tools available that can help you measure memory usage. For example, PyTorch has a memory profiler that you can use to see where memory is allocated.

Remember, these methods will only give you a rough estimate, and the actual memory usage can be higher due to various factors such as implementation details of the underlying libraries and overhead of the runtime environment.

I hope this helps! Let me know if you have any other questions. 😊

**User**

Hmm, I see, that's going nowhere then unfortunately. Okay, how about this, could we add something to our code to optimize the resource usage? You mentioned a PyTorch optimization feature as an example earlier. Could something like that be applied to our code?

**Bing**

Yes, there are several ways to optimize the resource usage of your code. Here are a few techniques that could be applied to your code:

1. **Gradient Checkpointing**: This is a technique for trading compute for memory. It can be particularly useful when training large models like BERT that have many layers. PyTorch provides a utility for gradient checkpointing that you can use. However, this is typically used during training rather than inference, and it might not be applicable in your case.

2. **TorchScript**: This is a way to serialize your PyTorch models, allowing them to run in a high-performance, non-Python environment. This could potentially reduce the memory footprint of your model.

3. **Quantization**: This involves converting the weights of your model from 32-bit floating point numbers to lower-precision formats, such as 8-bit integers. This can significantly reduce both the memory usage and computational requirements of your model. PyTorch provides utilities for both dynamic and static quantization.

4. **Pruning**: This involves removing some of the parameters in your model, such as the weights associated with certain neurons. This can reduce the memory requirements of your model, but it may also affect the model's accuracy.

5. **Onnx Runtime**: You can convert your PyTorch model to ONNX format and then use ONNX Runtime for inference. ONNX Runtime is designed to optimize and accelerate machine learning model inference across multiple hardware platforms.

Please note that these optimizations may require significant changes to your code and may also affect the accuracy of your model. They also often involve a trade-off between memory usage, computational requirements, and model accuracy. It's important to thoroughly test your model after applying these optimizations to ensure it still meets your requirements.

Here's an example of how you can apply quantization to your model:

```
import torch.quantization

# Quantize the model
quantized_model = torch.quantization.quantize_dynamic(
    model, {torch.nn.Linear}, dtype=torch.qint8
)

# Now you can use `quantized_model` instead of `model`
output = quantized_model(**input)
```

In this example, we're using dynamic quantization on the Linear layers of the model. Dynamic quantization is the simplest form of quantization, where the weights of the neurons are quantized from float32 to int8 at runtime.

I hope this helps! Let me know if you have any other questions. 😊

**User**

Alright, I will apply the code for quantization to our code now, thank you! Quick question about that: If I have an import torch line already, do I still need the import torch.quantization line?

**Bing**

Yes, you still need the `import torch.quantization` line. In Python, importing a module does not automatically import its submodules. So even if you have already imported `torch`, you still need to explicitly import `torch.quantization` to use the functions and classes it provides. Here's how you can do it:

```
import torch
import torch.quantization
```

Now, you can use the functions and classes from `torch.quantization` in your code. I hope this helps! Let me know if you have any other questions. 😊

**User**

Despite the optimization through quantization, there's still a risk that running it could freeze my device again due to too much resource usage, right? Is there anything we could add to the code to limit resource usage, for example only allowing it to use 12GB of RAM? And how about adding a kind of "emergency termination" that stops it after a certain amount of time if it hasn't finished by then?

**Bing**

Yes, even with optimization, running large models like finBERT can still be resource-intensive and there's a risk that it could use up a lot of your system's resources.

Unfortunately, Python doesn't provide a direct way to limit the amount of memory that a script can use. However, there are a few workarounds you could consider:

1. **Using a Memory Profiler**: Tools like `memory-profiler` can help you monitor your script's memory usage. You can use it to find out which parts of your code are consuming the most memory and try to optimize them.

2. **Running in a Docker Container**: Docker allows

### User

Despite the optimization through quantization, there's still a risk that running it could freeze my device again due to too much resource usage, right? Is there anything we could add to the code to limit resource usage, for example only allowing it to use 12GB of RAM? And how about adding a kind of "emergency termination" that stops it after a certain amount of time if it hasn't finished by then?

### Bing

Yes, even with optimization, running large models like finBERT can still be resource-intensive and there's a risk that it could use up a lot of your system's resources.

Unfortunately, Python doesn't provide a direct way to limit the amount of memory that a script can use. However, there are a few workarounds you could consider:

1. **Using a Memory Profiler**: Tools like `memory-profiler` can help you monitor your script's memory usage. You can use it to find out which parts of your code are consuming the most memory and try to optimize them.

2. **Running in a Docker Container**: Docker allows you to limit the amount of memory that a container (and therefore your script) can use. This would involve setting up Docker on your machine, creating a Dockerfile for your script, and running it in a Docker container.

As for adding an "emergency termination", you could consider using Python's `signal` module to set a timer and raise an exception if the script runs for too long. Here's a simple example:

```
import signal
import time

# Define your handler
def handler(signum, frame):
    raise Exception("Time is up!")

# Register the signal function handler
signal.signal(signal.SIGALRM, handler)

# Define a time limit
time_limit = 3600  # One hour, for example
signal.alarm(time_limit)

try:
    # Your code here...
    time.sleep(3600)  # This is just an example
except Exception as exc:
    print(exc)
```

In this example, if the code inside the `try` block runs for more than one hour, the `handler` function will raise an exception, effectively stopping the script.

Please note that these are advanced topics and might require some additional learning if you're new to Python. Also, these methods might not be applicable or effective in all situations or for all types of resource usage. It's important to thoroughly test any changes you make to ensure they're having the desired effect. 😊

### User

Alright, I've modified our code! Would you please look it over and check whether everything looks to be in order? I had to take out the comments due to the character limit of this prompt, sadly.

import torch import torch.quantization from lime.lime_text import LimeTextExplainer from transformers import AutoTokenizer, AutoModelForSequenceClassification import signal import time

def handler(signum, frame): raise Exception("Execution is taking too long due to too much resource usage and the script was stopped.") signal.signal(signal.SIGALRM, handler) time_limit = 3600 signal.alarm(time_limit) try: tokenizer = AutoTokenizer.from_pretrained("ProsusAI/finbert") model = AutoModelForSequenceClassification.from_pretrained("ProsusAI/finbert") input = tokenizer("Nvidia's stock more than quadrupled over the last 15 months as investors bought into Nvidia's market-leading position in artificial intelligence.", return_tensors="pt") quantized_model = torch.quantization.quantize_dynamic( model, {torch.nn.Linear}, dtype=torch.qint8 ) output = quantized_model(**input) probs = torch.nn.functional.softmax(output.logits, dim=-1) predicted_class = torch.argmax(probs) explainer = LimeTextExplainer(class_names=["Negative", "Neutral", "Positive"]) def predictor(texts): inputs = tokenizer(texts, return_tensors="pt", padding=True, truncation=True) outputs = model(**inputs) probs = torch.nn.functional.softmax(outputs.logits, dim=-1) return probs.detach().numpy() explanation = explainer.explain_instance("Nvidia's stock more than quadrupled over the last 15 months as investors bought into Nvidia's market-leading position in artificial intelligence.", predictor, num_features=6, labels=(predicted_class.item(),)) print(explanation) time.sleep(3600) except Exception as exc: print(exc)

### Bing

Your code looks mostly correct, but there are a couple of things you might want to adjust:

1. **Quantized Model Usage**: After quantizing the model, you correctly used the quantized model for getting the output. However, in the `predictor` function, you're still using the original `model` for getting the outputs. You should replace `model` with `quantized_model` in the `predictor` function.

2. **Sleep Function**: The `time.sleep(3600)` line will make your script pause for an hour. This line is not necessary and could be removed.

Here's how your adjusted code would look:

```
import torch
import torch.quantization
from lime.lime_text import LimeTextExplainer
from transformers import AutoTokenizer, AutoModelForSequenceClassification
import signal
import time

def handler(signum, frame):
    raise Exception("Execution is taking too long due to too much resource usage and the script was stopped.")
signal.signal(signal.SIGALRM, handler)
time_limit = 3600
signal.alarm(time_limit)
try:
    tokenizer = AutoTokenizer.from_pretrained("ProsusAI/finbert")
    model = AutoModelForSequenceClassification.from_pretrained("ProsusAI/finbert")
    input = tokenizer("Nvidia's stock more than quadrupled over the last 15 months as investors bought into Nvidia's market-leading position in artificial intell
    quantized_model = torch.quantization.quantize_dynamic(
        model, {torch.nn.Linear}, dtype=torch.qint8
    )
    output = quantized_model(**input)
    probs = torch.nn.functional.softmax(output.logits, dim=-1)
    predicted_class = torch.argmax(probs)
```

```
    explainer = LimeTextExplainer(class_names=["Negative", "Neutral", "Positive"])
    def predictor(texts):
        inputs = tokenizer(texts, return_tensors="pt", padding=True, truncation=True)
        outputs = quantized_model(**inputs)
        probs = torch.nn.functional.softmax(outputs.logits, dim=-1)
        return probs.detach().numpy()
    explanation = explainer.explain_instance("Nvidia's stock more than quadrupled over the last 15 months as investors bought into Nvidia's market-leading positi
                                     predictor,
                                     num_features=6,
                                     labels=(predicted_class.item(),))
    print(explanation)
except Exception as exc:
    print(exc)
```

I hope this helps! Let me know if you have any other questions. 😊

### User

Ah, thank you! I was wondering about the time.sleep function since that was part of what you showed me when I asked about the "emergency termination" feature. This is the part of the code I mean:

try: # Your code here… time.sleep(3600) # This is just an example except Exception as exc: print(exc)

Could you explain why you put the time.sleep(3600) line in there if it just pauses the script?

### Bing

I apologize for any confusion. The `time.sleep(3600)` line was included in the example to simulate a long-running operation. It's not necessary for your actual code.

In the context of the example, `time.sleep(3600)` makes the script pause for an hour. If the script takes longer than the specified `time_limit` to run, the `handler` function raises an exception to stop the script. This was just to demonstrate how the "emergency termination" feature would work.

In your actual code, you would replace `time.sleep(3600)` with the operations you want to perform (like loading the model, making predictions, etc.). If these operations take longer than the `time_limit`, the script will be stopped.

I hope this clears up the confusion! Let me know if you have any other questions. 😊

### User

Ah I see, no problem, I understand! I'll have to wait for access to better hardware until I can properly test our code. In the meantime, let's try and add some more cool stuff to it! For example, would it be possible to implement a simple GUI for it? I'm thinking it could have a text field where a user can enter their own text and then click a button to have the script use the user-entered text as input instead of the input being hardcoded in the script, if that could work?

### Bing

Absolutely, you can create a simple graphical user interface (GUI) for your script using a Python library like Tkinter. Here's a basic example of how you could do it:

```
import tkinter as tk
from tkinter import messagebox

def run_model():
    # Get the user's input from the text field
    user_input = text_field.get("1.0", "end-1c")
    if user_input:
        try:
            # Replace this line with the code to run your model
            # For example: result = run_finbert(user_input)
            result = "Model output for: " + user_input
            messagebox.showinfo("Result", result)
        except Exception as e:
            messagebox.showerror("Error", str(e))
    else:
        messagebox.showwarning("No Input", "Please enter some text.")

# Create a new Tkinter window
window = tk.Tk()

# Create a new text field
text_field = tk.Text(window, width=50, height=10)
text_field.pack()

# Create a new button that will run the model when clicked
run_button = tk.Button(window, text="Run Model", command=run_model)
run_button.pack()

# Start the Tkinter event loop
window.mainloop()
```

In this example, the GUI has a text field where the user can enter their own text, and a button that runs the model on the user's input when clicked. The `run_model` function is called when the button is clicked. It gets the user's input from the text field, runs the model on the input, and then shows a message box with the result.

You would need to replace the line `result = "Model output for: " + user_input` with the code to run your model. For example, you could define a function `run_finbert` that takes a string as input, runs your finBERT model on the input, and returns the result. Then you could call this function in `run_model`.

Please note that Tkinter is a basic GUI library and might not be suitable for more complex interfaces. If you need more advanced features, you might want to consider using a more powerful library like PyQt or Kivy.

I hope this helps! Let me know if you have any other questions. 😊

### User

Should the LIME-related portion of our code go between the run_model function block and before the Tkinter window creation line?

### Bing

The LIME-related portion of your code should be included within the `run_model` function. This is because you want to run the LIME explanation when the "Run Model" button is clicked. Here's how you can modify the `run_model` function to include the LIME explanation:

```
def run_model():
    # Get the user's input from the text field
    user_input = text_field.get("1.0", "end-1c")
    if user_input:
        try:
```

```
            # Tokenize the user's input
            input = tokenizer(user_input, return_tensors="pt")
            # Get the model's output
            output = quantized_model(**input)
            # Get the probabilities
            probs = torch.nn.functional.softmax(output.logits, dim=-1)
            # Get the predicted class
            predicted_class = torch.argmax(probs)
            # Define a prediction function for LIME
            def predictor(texts):
                inputs = tokenizer(texts, return_tensors="pt", padding=True, truncation=True)
                outputs = quantized_model(**inputs)
                probs = torch.nn.functional.softmax(outputs.logits, dim=-1)
                return probs.detach().numpy()
            # Run LIME explanation
            explanation = explainer.explain_instance(user_input,
                                                     predictor,
                                                     num_features=6,
                                                     labels=(predicted_class.item(),))
            # Show the explanation
            messagebox.showinfo("Result", str(explanation))
        except Exception as e:
            messagebox.showerror("Error", str(e))
    else:
        messagebox.showwarning("No Input", "Please enter some text.")
```

In this modified version of `run_model`, the user's input is tokenized and then fed into the model when the "Run Model" button is clicked. The LIME explanation is then generated based on the model's output. The explanation is displayed in a message box.

I hope this helps! Let me know if you have any other questions. 😊

### User

Should the "emergency termination" part of our code (which is currently around the model running part) also go inside the run_model function? So it would be structured as the run_model function around the "emergency termination" function which is around the code part that runs finBERT and LIME?

### Bing

Yes, you're correct. The "emergency termination" part of the code should be placed inside the `run_model` function. This way, the timeout will apply to the model running and LIME explanation generation process which are triggered when the "Run Model" button is clicked. Here's how you can structure it:

```
def run_model():
    # Get the user's input from the text field
    user_input = text_field.get("1.0", "end-1c")
    if user_input:
        try:
            signal.signal(signal.SIGALRM, handler)
            signal.alarm(time_limit)
            try:
                # Tokenize the user's input
                input = tokenizer(user_input, return_tensors="pt")
                # Get the model's output
                output = quantized_model(**input)
                # Get the probabilities
                probs = torch.nn.functional.softmax(output.logits, dim=-1)
                # Get the predicted class
                predicted_class = torch.argmax(probs)
                # Define a prediction function for LIME
                def predictor(texts):
                    inputs = tokenizer(texts, return_tensors="pt", padding=True, truncation=True)
                    outputs = quantized_model(**inputs)
                    probs = torch.nn.functional.softmax(outputs.logits, dim=-1)
                    return probs.detach().numpy()
                # Run LIME explanation
                explanation = explainer.explain_instance(user_input,
                                                         predictor,
                                                         num_features=6,
                                                         labels=(predicted_class.item(),))
                # Show the explanation
                messagebox.showinfo("Result", str(explanation))
            except Exception as exc:
                print(exc)
        except Exception as e:
            messagebox.showerror("Error", str(e))
    else:
        messagebox.showwarning("No Input", "Please enter some text.")
```

In this adjusted version of `run_model`, the "emergency termination" is set right before the model running and LIME explanation generation process. If these operations take longer than the `time_limit`, the script will be stopped.

I hope this helps! Let me know if you have any other questions. 😊

### User

So we have a Tkinter window in which the user enters their input and then presses a button to run the model and LIME explainer with their input. I see in the code that there is also functionality for displaying the result. Does the result get displayed in the same Tkinter window or does the script create a new window to display the result?

### Bing

The result gets displayed in a separate dialog box, not in the same Tkinter window. The `messagebox.showinfo("Result", str(explanation))` line in the code creates a new dialog box that displays the result. The first argument to `showinfo` is the title of the dialog box, and the second argument is the message to display.

If you want to display the result in the same Tkinter window, you could use a Label or Text widget to display the result. Here's an example of how you could modify the `run_model` function to display the result in the same window:

```
def run_model():
    # Get the user's input from the text field
    user_input = text_field.get("1.0", "end-1c")
    if user_input:
        try:
            signal.signal(signal.SIGALRM, handler)
            signal.alarm(time_limit)
            try:
                # Tokenize the user's input
                input = tokenizer(user_input, return_tensors="pt")
                # Get the model's output
```

```
            output = quantized_model(**input)
            # Get the probabilities
            probs = torch.nn.functional.softmax(output.logits, dim=-1)
            # Get the predicted class
            predicted_class = torch.argmax(probs)
            # Define a prediction function for LIME
            def predictor(texts):
                inputs = tokenizer(texts, return_tensors="pt", padding=True, truncation=True)
                outputs = quantized_model(**inputs)
                probs = torch.nn.functional.softmax(outputs.logits, dim=-1)
                return probs.detach().numpy()
            # Run LIME explanation
            explanation = explainer.explain_instance(user_input,
                                                     predictor,
                                                     num_features=6,
                                                     labels=(predicted_class.item(),))
            # Display the explanation in the same window
            result_label.config(text=str(explanation))
        except Exception as exc:
            print(exc)
    except Exception as e:
        messagebox.showerror("Error", str(e))
else:
    messagebox.showwarning("No Input", "Please enter some text.")

# Create a new label to display the result
result_label = tk.Label(window, text="")
result_label.pack()
```

In this modified version of `run_model`, a Label widget is used to

### User

So we have a Tkinter window in which the user enters their input and then presses a button to run the model and LIME explainer with their input. I see in the code that there is also functionality for displaying the result. Does the result get displayed in the same Tkinter window or does the script create a new window to display the result?

### Bing

The result is displayed in a separate dialog box, not in the same Tkinter window. The line `messagebox.showinfo("Result", str(explanation))` in the code creates a new dialog box that displays the result. The first argument to `showinfo` is the title of the dialog box, and the second argument is the message to display.

If you want to display the result in the same Tkinter window, you could use a Label or Text widget to display the result. Here's an example of how you could modify the `run_model` function to display the result in the same window:

````
def run_model():
    # Get the user's input from the text field
    user_input = text_field.get("1.0", "end-1c")
    if user_input:
        try:
            signal.signal(signal.SIGALRM, handler)
            signal.alarm(time_limit)
            try:
                # Tokenize the user's input
                input = tokenizer(user_input, return_tensors="pt")
                # Get the model's output
                output = quantized_model(**input)
                # Get the probabilities
                probs = torch.nn.functional.softmax(output.logits, dim=-1)
                # Get the predicted class
                predicted_class = torch.argmax(probs)
                # Define a prediction function for LIME
                def predictor(texts):
                    inputs = tokenizer(texts, return_tensors="pt", padding=True, truncation=True)
                    outputs = quantized_model(**inputs)
                    probs = torch.nn.functional.softmax(outputs.logits, dim=-1)
                    return probs.detach().numpy()
                # Run LIME explanation
                explanation = explainer.explain_instance(user_input,


<div class="user-title">User</div>

Oh, sorry, but it seems your message got cut off. Could you please repeat it? Also feel free to leave out the inner parts of the code that concern the "emergency


<div class="bot-title">Bing</div>

Of course, I apologize for the confusion. Here's a simplified version of the code with a Tkinter GUI:

```python
import tkinter as tk
from tkinter import messagebox

def run_model():
    # Get the user's input from the text field
    user_input = text_field.get("1.0", "end-1c")
    if user_input:
        try:
            # Your code to run the model and LIME explainer goes here
            # ...
            # Show the explanation in the same window
            result_label.config(text=str(explanation))
        except Exception as e:
            messagebox.showerror("Error", str(e))
    else:
        messagebox.showwarning("No Input", "Please enter some text.")

# Create a new Tkinter window
window = tk.Tk()

# Create a new text field
text_field = tk.Text(window, width=50, height=10)
text_field.pack()

# Create a new label to display the result
result_label = tk.Label(window, text="")
result_label.pack()

# Create a new button that will run the model when clicked
run_button = tk.Button(window, text="Run Model", command=run_model)
````
````

```
run_button.pack()

# Start the Tkinter event loop
window.mainloop()
```

In this version, the `run_model` function gets the user's input from the text field, runs your model and LIME explainer on the input (you'll need to add this part), and then displays the result in the same window using a Label widget. The result is updated every time the "Run Model" button is clicked.

I hope this helps! Let me know if you have any other questions. 😊

### User

What exactly is the result that gets displayed? Or rather, what does it look like? I've seen on several websites that LIME can display its result (the explanation) in an image format, but I'm assuming in the case of our code here it'll be formatted text?

### Bing

Yes, you're correct. In the case of your code, the result displayed is a text-based explanation. The `explain_instance` method of LIME's `TextExplainer` returns an `Explanation` object, which includes information about the explanation. When you convert this object to a string (with `str(explanation)`), it generates a text-based summary of the explanation.

This summary includes the predicted class and a list of the features (words or phrases in your text) that contribute most to the prediction, along with their weights. The features are listed in order of their weights, so the first feature in the list is the one that contributes most to the prediction.

While LIME can indeed generate visual explanations for image data, for text data the explanations are typically presented as lists of features (words or phrases) and their weights. However, you could certainly create a visual representation of this data if you wanted to, such as a bar chart showing the weights of the top features. This would require additional coding and use of a library like matplotlib.

I hope this helps! Let me know if you have any other questions. 😊

### User

Check out this line of code I found on a website's example LIME code: [exp.as_pyplot_figure(label=1) for exp in sp_obj.sp_explanations] Apparently this leads to a visualization of the text data explanations? Though it doesn't look like Python code to me, or is it?

### Bing

Yes, that is indeed Python code! It's using a list comprehension, which is a compact way of creating a new list by iterating over an existing one and applying a function to each element.

Here's a breakdown of what this line of code is doing:

- `sp_obj.sp_explanations` is a list of `Explanation` objects. Each `Explanation` object represents a LIME explanation for a single instance.
- `for exp in sp_obj.sp_explanations` iterates over each `Explanation` object in the list.
- `exp.as_pyplot_figure(label=1)` generates a pyplot figure for the explanation. The `label=1` argument specifies that it should generate the figure for the class with label 1.
- The list comprehension `[exp.as_pyplot_figure(label=1) for exp in sp_obj.sp_explanations]` generates a pyplot figure for each explanation in the list.

So, this line of code is generating pyplot figures for all the explanations in `sp_obj.sp_explanations`. These figures provide a visual representation of the explanations, which can be helpful for understanding them.

However, please note that this line of code doesn't actually display the figures. To display them, you would typically use `matplotlib.pyplot.show()` after generating the figures.

I hope this helps! Let me know if you have any other questions. 😊

### User

Oh okay, I see, thanks for explaining! Then let's try adding another element to our script's Tkinter window (which so far contains the user input text box, a button to run the model + LIME explainer, and a label widget to display the result). Can we add an element or widget or something below the result-displaying label widget that can display a graphical visualization of our result? We'll probably have to write a function to generate the visualization first though, as you mentioned, with something like matplotlib.

### Bing

Yes, you can certainly add a graphical visualization to your Tkinter window. However, Tkinter's built-in widgets don't support complex graphics like plots out of the box. You'll need to use a library like `matplotlib` to create the plot, and then display it in a Tkinter canvas or label.

Here's a basic example of how you could modify your code to include a plot:

```python
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

def run_model():
    # Get the user's input from the text field
    user_input = text_field.get("1.0", "end-1c")
    if user_input:
        try:
            # Your code to run the model and LIME explainer goes here
            # ...
            # Show the explanation in the same window
            result_label.config(text=str(explanation))

            # Create a new figure
            fig = plt.Figure(figsize=(5, 4), dpi=100)
            ax = fig.add_subplot(111)

            # Your code to generate the plot goes here
            # For example:
            ax.plot([1, 2, 3, 4], [1, 4, 2, 3])

            # Create a new canvas and draw the plot onto it
            canvas = FigureCanvasTkAgg(fig, master=window)
            canvas.draw()

            # Add the canvas to the Tkinter window
            canvas.get_tk_widget().pack()

        except Exception as e:
            messagebox.showerror("Error", str(e))
    else:
        messagebox.showwarning("No Input", "Please enter some text.")
```

```
# Your existing Tkinter code goes here
# ...
```

In this example, a new `Figure` is created using `matplotlib.pyplot`. A subplot is added to the figure, and then a plot is drawn on the subplot. The plot is then drawn onto a new `FigureCanvasTkAgg` canvas, which is added to the Tkinter window.

You would need to replace the line `ax.plot([1, 2, 3, 4], [1, 4, 2, 3])` with the code to generate your own plot. This could involve plotting the weights of the features from the LIME explanation, for example.

Please note that this is a basic example and might need to be adjusted based on your specific needs. Also, you'll need to install the `matplotlib` package if you haven't done so already. You can do this by running `pip install matplotlib`.

I hope this helps! Let me know if you have any other questions. 😊

### User

That helps a lot, thank you! Could you please elaborate on how I could go about plotting the weights of my LIME explanation's features? Right now what I can see in our existing code is that we define a number of features in this part:

explanation = explainer.explain_instance(user_input, predictor, num_features=6, labels=(predicted_class.item(),))

However, I'm not sure where to get the names of our features and I think we need those for the plotting, right?

### Bing

Yes, you're correct. The features in the context of LIME for text data are typically the words or phrases in your input text. The `explain_instance` method returns an `Explanation` object, which includes information about the explanation. You can get the names and weights of the features from this object.

The `Explanation` object has a method called `as_list`, which returns a list of tuples. Each tuple represents a feature and its weight. The first element of the tuple is the feature name (a word or phrase from your input text), and the second element is the weight of the feature.

Here's how you can modify your code to get the feature names and weights, and plot them:

```
def run_model():
    # Get the user's input from the text field
    user_input = text_field.get("1.0", "end-1c")
    if user_input:
        try:
            # Your code to run the model and LIME explainer goes here
            # ...
            # Show the explanation in the same window
            result_label.config(text=str(explanation))

            # Get the features and their weights
            features = explanation.as_list()

            # Separate the feature names and weights
            feature_names = [feature[0] for feature in features]
            feature_weights = [feature[1] for feature in features]

            # Create a new figure
            fig = plt.Figure(figsize=(5, 4), dpi=100)
            ax = fig.add_subplot(111)

            # Plot the feature weights
            ax.bar(feature_names, feature_weights)

            # Create a new canvas and draw the plot onto it
            canvas = FigureCanvasTkAgg(fig, master=window)
            canvas.draw()

            # Add the canvas to the Tkinter window
            canvas.get_tk_widget().pack()

        except Exception as e:
            messagebox.showerror("Error", str(e))
    else:
        messagebox.showwarning("No Input", "Please enter some text.")
```

In this modified version of `run_model`, the feature names and weights are extracted from the `Explanation` object and plotted as a bar chart. The chart is then displayed in the same Tkinter window.

I hope this helps! Let me know if you have any other questions. 😊

### User

Okay, I see, thank you! Unfortunately I feel like I got a bit lost in all the code, so to speak. Let's take a step back for a moment and look at what our result here actually is. What we do so far in the code is this: We run the finBERT model (which classifies a given user input as positive, neutral or negative) and a LIME text explainer, which is supposed to explain to us which parts of the given user input lead to finBERT's classification. To prevent our device from freezing endlessly in case of too much resource usage, we implemented an "emergency termination" feature which stops the script after a set amount of time if no result has been achieved. We added a GUI in which the user can input their text and which shows the output in the same GUI window. Right now we're trying to add an element in the same GUI window which will display a graphical representation of the LIME explainer result, which would be a matplotlib plot showing how much a word in the user input text contributes to finBERT's classification of the overall user input text. Does that sound right, am I up to speed here?

### Bing

Yes, you've got it! You've accurately summarized the functionality of your script. Here's a recap:

1. **finBERT Model**: The script uses the finBERT model to classify a given user input as positive, neutral, or negative. This is a sentiment analysis task.
2. **LIME Text Explainer**: The script uses LIME (Local Interpretable Model-Agnostic Explanations) to explain the classification made by the finBERT model. It identifies which parts (words or phrases) of the user input contribute most to the classification.
3. **Emergency Termination**: To prevent excessive resource usage that could freeze your device, the script includes a feature that stops the script if it runs for longer than a set amount of time without producing a result.
4. **GUI**: The script includes a graphical user interface (GUI) where the user can enter their text. The GUI displays the output (the classification and the LIME explanation) in the same window.
5. **Visualization**: You're currently working on adding a feature to the GUI that will display a graphical representation of the LIME explanation. This will be a plot showing how much each word in the user input text contributes to the finBERT classification.

Your understanding is correct! If you have any more questions or need further clarification, feel free to ask. 😊

**User**

Awesome, thank you! Could you please clarify real quick what the resulting plot would look like? Would it be just dots or a bar chart or something else?

**Bing**

The resulting plot would be a bar chart. Each bar in the chart represents a feature (a word or phrase from your input text). The length of the bar corresponds to the weight of the feature, which indicates how much it contributes to the model's prediction.

The features are plotted on the x-axis, and their weights are plotted on the y-axis. The direction of the bar (positive or negative on the y-axis) indicates whether the feature contributes towards a positive or negative classification.

This type of plot provides a visual way to understand which words or phrases in your text are most important for the model's prediction. It can be especially useful when working with complex models like BERT, where it's not immediately obvious how the model is making its decisions.

I hope this helps! Let me know if you have any other questions. 😊

**User**

Alright, I have implemented the code for visualization! I'm thinking about what else we could add, but the script seems pretty complete to me now. My primary goal was to understand how to apply LIME to an AI model by setting up an implementation of it myself, which I've achieved thanks to your great help. A secondary goal was to achieve a kind of visual demonstration which I can show to others as a practical and approachable example of explainable AI - would you say that we have achieved that? Or do you think there's something else we could or should add in order to improve this script as an approachable demo of a possible application of explainable AI?

**Bing**

I'm glad to hear that you've found our discussion helpful! Based on your description, it sounds like you've achieved your primary goal of understanding how to apply LIME to an AI model. You've also made significant progress towards your secondary goal of creating a visual demonstration of explainable AI.

Your script, which uses the finBERT model to classify text and LIME to explain the classifications, is a great example of explainable AI in action. The addition of a GUI makes the script more interactive and user-friendly, and the graphical visualization of the LIME explanation helps make the explanation more understandable and approachable.

As for potential improvements, here are a few suggestions:

1. **Interactivity**: You could add more interactivity to the GUI. For example, you could allow users to adjust the number of features used in the LIME explanation, or to choose the class for which the explanation is generated.

2. **Error Handling**: You could add more robust error handling to the script. For example, you could catch and handle specific exceptions that might be raised during the execution of the model or the LIME explainer.

3. **Documentation**: You could add more comments to the script explaining what each part of the code does. This would make the script easier to understand for others who might want to learn from it or modify it.

4. **Performance Monitoring**: You could add code to monitor the performance of the script, such as the time taken to run the model and generate the explanation, or the memory usage of the script.

Remember, the best way to improve a script like this is to use it, gather feedback, and make iterative improvements. I hope this helps, and I'm here if you have any other questions! 😊

**User**

Oh, your 4th point about performance monitoring sounds like something we could really use, considering the resource usage issues we had that led us to set up the "emergency termination" feature in the first place! Time taken to run and memory usage are good metrics to have, but is there by chance also a way to calculate/predict/estimate in advance how long the script will take to achieve a result with the resources available to it?

**Bing**

Predicting the exact execution time of a script in advance can be quite challenging due to the dynamic nature of many factors such as system load, available resources, and the specific operations being performed by the script. However, you can estimate the time complexity of your code in terms of Big O notation, which gives you an idea of how the execution time will grow as the size of the input increases.

For measuring the execution time of your script, Python provides several modules like `time`, `timeit`, and `datetime`. Here's a simple example of how you can measure the execution time of your script using the `time` module:

```
import time

start_time = time.time()

# Your code here...

end_time = time.time()
elapsed_time = end_time - start_time
print(f"Execution time: {elapsed_time} seconds")
```

In this code, `time.time()` returns the current time in seconds since the epoch (a fixed point in time, usually January 1, 1970). By calling this function before and after the code you want to time, you can calculate the elapsed time, which is the execution time of your code.

As for memory usage, there are several Python libraries that can help you measure this, such as `memory-profiler` and `psutil`.

Please note that these methods provide measurements and estimates, but they don't allow you to predict the exact execution time or memory usage in advance. The actual execution time and memory usage can depend on many factors, including the specific inputs to your script, the state of your system, and the behavior of other processes running on your system.

I hope this helps! Let me know if you have any other questions. 😊

**User**

Okay Bing, unfortunately we're about to hit the message number limit for this conversation :( I would very much like to continue this conversation in a new one though so that we can continue to improve on our script! Would you please summarize, for yourself, all the context that you need from this entire conversation so that we can start where we left off here in a new conversation?

**Bing**

Absolutely, I'd be happy to summarize our conversation for future reference:

1. **Goal**: Your primary goal was to understand how to apply LIME (Local Interpretable Model-Agnostic Explanations) to an AI model, specifically the finBERT model for sentiment analysis. Your secondary goal was to create a visual demonstration of explainable AI that you can show to others as a practical and approachable example.

2. **Script**: We've been working on a Python script that uses the finBERT model to classify a given user input as positive, neutral, or negative. It also uses LIME to explain the classification, identifying which parts of the user input contribute most to the classification.

3. **Emergency Termination**: We added a feature to the script that stops it if it runs for longer than a set amount of time without producing a result. This was implemented to prevent excessive resource usage that could freeze your device.

4. **GUI**: We added a graphical user interface (GUI) to the script using Tkinter. The GUI allows the user to enter their text and displays the output (the classification and the LIME explanation) in the same window.

5. **Visualization**: We're currently working on adding a feature to the GUI that will display a graphical representation of the LIME explanation. This will be a plot showing how much each word in the user input text contributes to the finBERT classification.

6. **Performance Monitoring**: We discussed adding code to monitor the performance of the script, such as the time taken to run the model and generate the explanation, or the memory usage of the script.

I hope this summary is helpful! I look forward to continuing our conversation in a new thread. 😊