Shapley Values _OpenAI-OSS-20B

https://youtu.be/y901kYsI5O8?si=IrAqpcZdFYUFACu9

💡 The **GPT-OSS-20B** model is an **open-weight** model released by OpenAI, and its weights are **freely available for download** and use under the **Apache 2.0 license**.

Primary Access Points

1. **Hugging Face Hub:** The official Hugging Face repository provides the model weights. You can download them directly using the Hugging Face CLI:

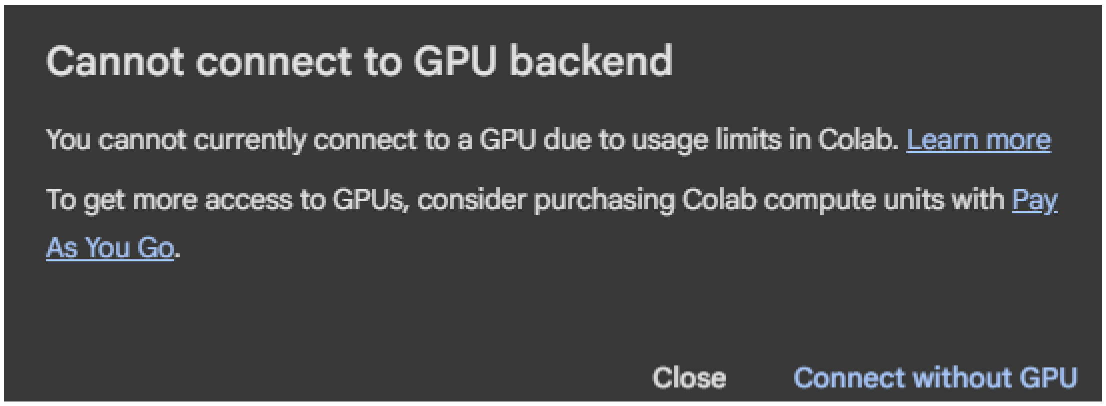
```
huggingface-cli download openai/gpt-oss-20b --include "original/*" --local-dir gpt-oss-20b/
```

2. **GitHub Repository:** OpenAI provides a **GitHub repository** for `gpt-oss` which includes reference inference code and resources.
3. **OpenAI Cookbook:** The official OpenAI Cookbook has step-by-step guides for setup with various supported runtimes.

💡 **Using `FastLanguageModel.from_pretrained(...)` from Unsloth, you are loading the model weights locally. You can run inference entirely on your machine.**

Code:

Drawback of below code: 👇


Cannot connect to GPU backend

You cannot currently connect to a GPU due to usage limits in Colab. Learn more

To get more access to GPUs, consider purchasing Colab compute units with Pay As You Go.

```
%%capture
!pip install --upgrade -qqq uv
try: import numpy; get_numpy = f"numpy=={numpy.__version__}"
except: get_numpy = "numpy"
!uv pip install -qqq \
    "torch>=2.8.0" "triton>=3.4.0" {get_numpy} torchvision bitsandbytes "trans
formers>=4.55.3" \
    "unsloth_zoo[base] @ git+https://github.com/unslothai/unsloth-zoo" \
    "unsloth[base] @ git+https://github.com/unslothai/unsloth" \
    git+https://github.com/triton-lang/triton.git@05b2c186c1b6c9a08375389d5
efe9cb4c401c075#subdirectory=python/triton_kernels
!uv pip install --upgrade --no-deps transformers==4.56.2 tokenizers
!uv pip install --no-deps trl==0.22.2
```

Installing dependencies:

All those `uv pip install` lines are just installing:

- **torch** → main computation framework
- **bitsandbytes** → enables quantization (run big models on small GPUs)
- **transformers** → model architecture & tokenizer
- **unsloth** and **unsloth_zoo** → optimized models + loader
- **triton** → speeds up matrix math

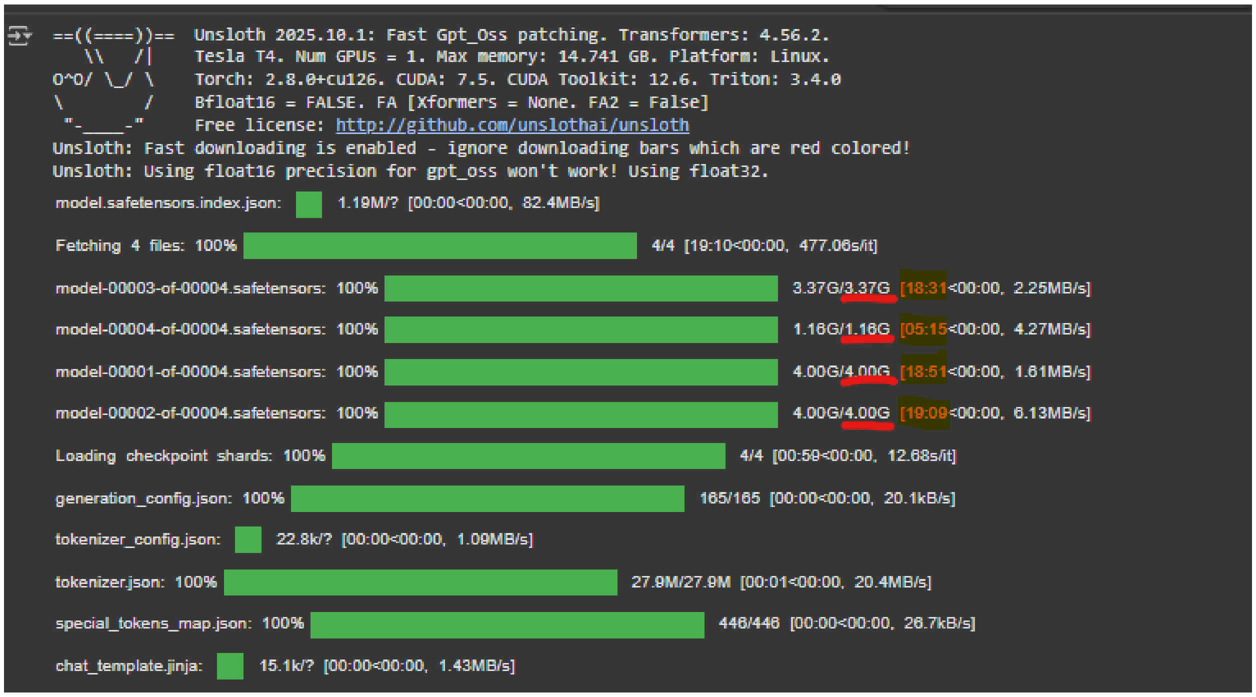
- **trl** → for reinforcement learning (used for fine-tuning later)

`%%capture` just hides the installation output (so your Colab cell doesn't flood).

```
%%capture
!uv pip install --force-reinstall --no-deps git+https://github.com/unslothai/unsl
oth-zoo
!uv pip install --force-reinstall --no-deps git+https://github.com/unslothai/unsl
oth
```

1️⃣ Access the weights

```
import torch
from unsloth import FastLanguageModel

# ----------------------
# Load model locally
# ----------------------
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/gpt-oss-20b-unsloth-bnb-4bit",
    dtype=None,          # auto detect
    load_in_4bit=True, # Changed to True to load in 4-bit quantization
    attn_implementation="eager"# FP16 for logprobs
)
```



```
==((====))==  Unsloth 2025.10.1: Fast Gpt_Oss patching. Transformers: 4.56.2.
   \\   /|    Tesla T4. Num GPUs = 1. Max memory: 14.741 GB. Platform: Linux.
O^O/ \_/ \    Torch: 2.8.0+cu126. CUDA: 7.5. CUDA Toolkit: 12.6. Triton: 3.4.0
\        /    Bfloat16 = FALSE. FA [Xformers = None. FA2 = False]
 "-____-"     Free license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are red colored!
Unsloth: Using float16 precision for gpt_oss won't work! Using float32.
model.safetensors.index.json:  1.19M/? [00:00<00:00, 82.4MB/s]
Fetching 4 files: 100%  4/4 [19:10<00:00, 477.06s/it]
model-00003-of-00004.safetensors: 100%  3.37G/3.37G [18:31<00:00, 2.25MB/s]
model-00004-of-00004.safetensors: 100%  1.16G/1.16G [05:15<00:00, 4.27MB/s]
model-00001-of-00004.safetensors: 100%  4.00G/4.00G [18:51<00:00, 1.61MB/s]
model-00002-of-00004.safetensors: 100%  4.00G/4.00G [19:09<00:00, 6.13MB/s]
Loading checkpoint shards: 100%  4/4 [00:59<00:00, 12.68s/it]
generation_config.json: 100%  165/165 [00:00<00:00, 20.1kB/s]
tokenizer_config.json:  22.8k/? [00:00<00:00, 1.09MB/s]
tokenizer.json: 100%  27.9M/27.9M [00:01<00:00, 20.4MB/s]
special_tokens_map.json: 100%  446/446 [00:00<00:00, 26.7kB/s]
chat_template.jinja:  15.1k/? [00:00<00:00, 1.43MB/s]
```

`model_name = "unsloth/gpt-oss-20b-unsloth-bnb-4bit"` **→ This means the original 20B parameter model (which would normally need ~40–80 GB of VRAM) is compressed to 4 bits per weight instead of 16 or 32.**

✅ Impact of 4-bit quantization:

Aspect	Effect
Model size	~4× smaller (so a 20B model can fit on 16GB GPU)
Speed	Faster load and inference
Accuracy	Slight drop (1–3%), because some precision is lost
Log probabilities	Slightly less precise, because quantization introduces rounding error

This actually *loads* the model:

- It downloads it from Hugging Face
- Uses the **Unsloth-optimized** loading method
- `load_in_4bit=False` means "don't quantize again," because the model is already pre-quantized
- `tokenizer` handles converting text → tokens → IDs

✨ `attn_implementation="eager"`

This flag controls **how attention operations are computed** inside the Transformer:

Option	Meaning	When to use
`"eager"`	Uses normal PyTorch operations (safe, portable)	Default in Unsloth / Transformers
`"flash_attention_2"` or `"sdpa"`	Uses fused CUDA kernels for faster attention	Use when your GPU supports Flash Attention

👉 Setting `"eager"` makes sure everything works even on older GPUs like Colab T4, but it's slightly slower.

Here, the model weights are downloaded and cached locally.

> ✅ So, weights are accessible, but note: you can't modify the 4-bit quantized weights directly unless you set **`load_in_4bit`** =False and have enough memory (~40–50 GB for full 20B FP16)

2️⃣ Computing log probabilities

Log probabilities give you how confident the model is about each token in the output. For a prompt + target output:

1. Encode prompt → get `input_ids`.
2. Encode target text → get `target_ids`.
3. Pass concatenated IDs through the model to get logits.
4. Compute `log_softmax` over logits.
5. Gather log probabilities corresponding to **actual target tokens**.

```
import torch
from unsloth import FastLanguageModel

# ----------------------
# Load model locally
# ----------------------
```

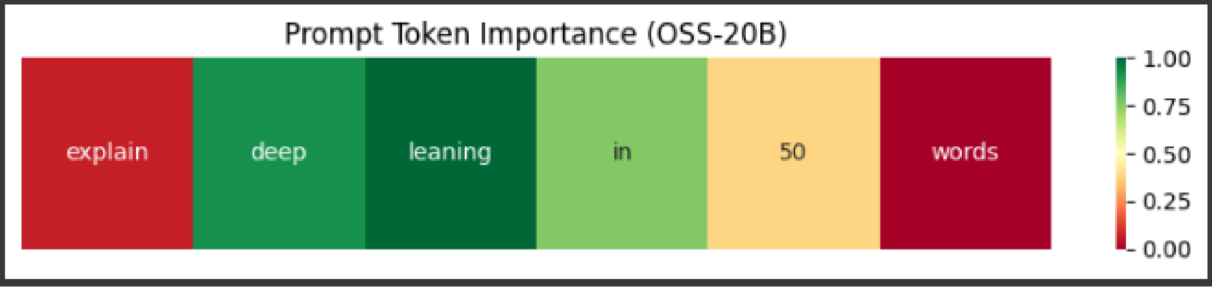
```
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/gpt-oss-20b-unsloth-bnb-4bit",
    dtype=None,          # auto detect
    load_in_4bit=False,
    attn_implementation="eager"# FP16 for logprobs
)
```

💡 **‼️The code can take more than 30-40 Mins to run.**

```
import torch
from unsloth import FastLanguageModel
import numpy as np # Import numpy

# -----------------------------
# Inputs
# -----------------------------
prompt = "explain deep leaning in 50 words"
target_output = ("""
Deep learning is a subset of machine learning that uses multi-layered neural
networks to learn from vast amounts of data. Inspired by the human brain's st
ructure, these
networks automatically identify patterns and features, enabling them to perfor
m complex tasks
like image recognition, natural language processing, and autonomous driving
"""
)
# prompt_tokens = prompt.split() # Keep original prompt for heatmap
target_ids = tokenizer(target_output).input_ids

# -----------------------------
# Function: compute log probabilities
# ------------------------------
def compute_logprobs(prompt, target_text):
    """Return total log probability of target_text given the prompt."""
    inputs = tokenizer.apply_chat_template(
        [{"role": "user", "content": prompt}],
        add_generation_prompt=True,
        return_tensors="pt",
        return_dict=True
    ).to(model.device)

    target_input_ids = tokenizer(target_text).input_ids
    all_input_ids = torch.cat([inputs.input_ids, torch.tensor([target_input_ids], d
evice=model.device)], dim=1)

    with torch.no_grad():
        outputs = model(all_input_ids)
        logits = outputs.logits  # shape: [1, seq_len, vocab_size]

    # Only consider target tokens
    shift_logits = logits[:, -len(target_input_ids)-1:-1, :]
    shift_labels = torch.tensor([target_input_ids], device=model.device)

    log_probs = torch.nn.functional.log_softmax(shift_logits, dim=-1)
    token_log_probs = log_probs.gather(2, shift_labels.unsqueeze(-1)).squeeze
(-1)  # [1, tgt_len]

    return token_log_probs.squeeze(0).cpu().numpy(), float(token_log_probs.su
m())

# ------------------------------
# Baseline
# ------------------------------
baseline_token_probs, baseline_total_logprob = compute_logprobs(prompt, ta
rget_output)
print(f"Baseline total log probability: {baseline_total_logprob:.3f}")
```

```
# -----------------------------
# Leave-One-Out: prompt token importance
# -----------------------------
importance_scores = []
prompt_tokens = prompt.split() # Define prompt_tokens here for the loop
for i, tok in enumerate(prompt_tokens):
    modified_prompt = " ".join(prompt_tokens[:i] + prompt_tokens[i+1:])
    _, total_lp = compute_logprobs(modified_prompt, target_output)
    importance = baseline_total_logprob - total_lp
    importance_scores.append(importance)

# Normalize to 0–1
importance_scores = np.array(importance_scores)
importance_scores = (importance_scores - importance_scores.min()) / (np.ptp
(importance_scores) + 1e-8)

# -----------------------------
# Print token importance
# -----------------------------
print("\nToken Importance Scores:")
for tok, score in zip(prompt_tokens, importance_scores):
    print(f"{tok:<15} → {score:.3f}")

# -----------------------------
# Heatmap visualization
# -----------------------------
import seaborn as sns # Import seaborn
import matplotlib.pyplot as plt # Import matplotlib

plt.figure(figsize=(max(10, len(prompt_tokens)), 1.5))
sns.heatmap([importance_scores], annot=[prompt_tokens], fmt="", cmap="Rd
YlGn", cbar=True, xticklabels=False, yticklabels=False)
plt.title("Prompt Token Importance (OSS-20B)")
plt.show()
```

```
Baseline total log probability: -738.000

Token Importance Scores:
explain          -> 0.065
deep             -> 0.912
leaning          -> 1.000
in               -> 0.756
50               -> 0.382
words            -> 0.000
```



🌟Shapley values with OpenAI-OSS-20B Model

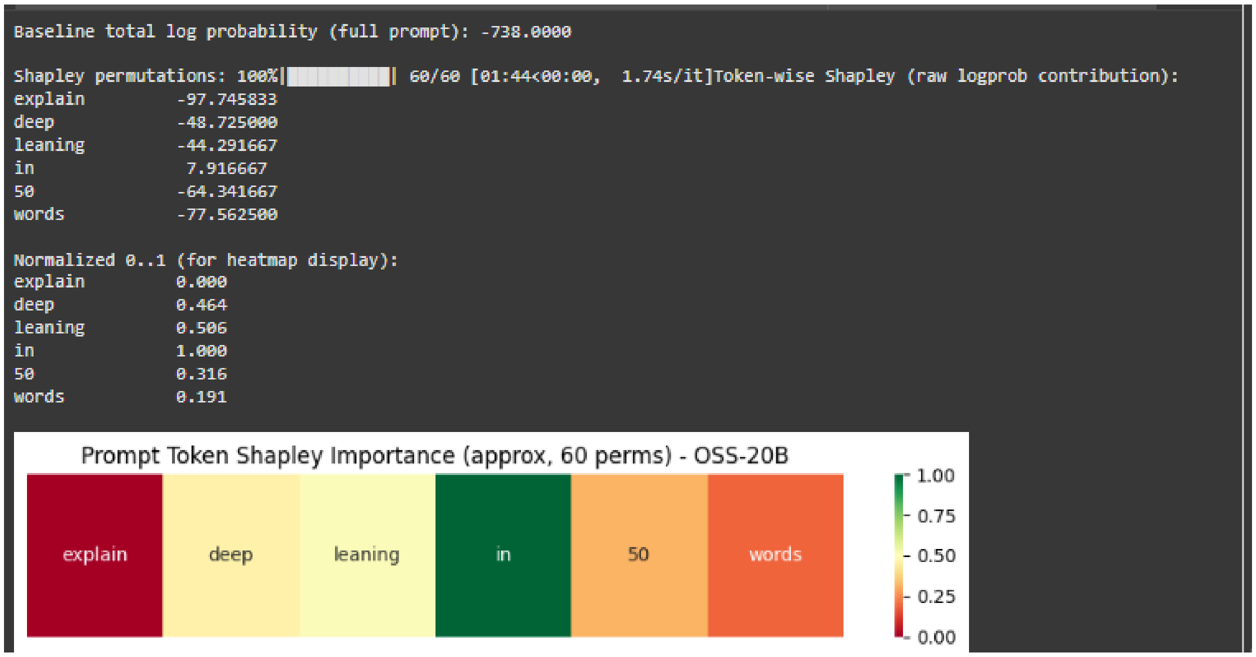
```
# Shapley approx for prompt token importance using OSS-20B (Unsloth)
# Requires: unsloth model + tokenizer loaded as `model, tokenizer`
# If you haven't loaded, uncomment the load block and set model_name accor
dingly.

import torch
import numpy as np
from tqdm import tqdm
import random
import seaborn as sns
import matplotlib.pyplot as plt

# ---------------------------
# Optional: load model/tokenizer (uncomment if not already loaded)
# ---------------------------
# from unsloth import FastLanguageModel
# model_name = "unsloth/gpt-oss-20b-unsloth-bnb-4bit"
# model, tokenizer = FastLanguageModel.from_pretrained(
#     model_name=model_name,
#     dtype=None,
#     max_seq_length=4096,
#     load_in_4bit=False,
#     full_finetuning=False
# )
# if torch.cuda.is_available():
#     model = model.to("cuda")
# model.eval()

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

# ---------------------------
# User prompt + target (kept same as your provided example)
# ---------------------------
prompt = "explain deep leaning in 50 words"
target_output = ("""
Deep learning is a subset of machine learning that uses multi-layered neural
networks to learn from vast amounts of data. Inspired by the human brain's st
ructure, these
networks automatically identify patterns and features, enabling them to perfor
m complex tasks
like image recognition, natural language processing, and autonomous driving
""")

# Work at whitespace token granularity to stay consistent with your previous c
ode
prompt_tokens = prompt.split()

# ---------------------------
# Reuse compute_logprobs function (returns per-token logprobs and total)
# ---------------------------
def compute_logprobs(prompt_text, target_text):
    """
    Returns (token_logprob_array, total_logprob)
    - token_logprob_array: numpy array of shape (tgt_len,) with log-probs for e
```

```
ach target token
    - total_logprob: float sum of token log-probs
    """
    # Build model-specific input using apply_chat_template (same pattern as yo
ur code)
    inputs = tokenizer.apply_chat_template(
        [{"role": "user", "content": prompt_text}],
        add_generation_prompt=True,
        return_tensors="pt",
        return_dict=True
    ).to(model.device)

    tgt_ids = tokenizer(target_text).input_ids
    # cat into a single tensor
    all_input_ids = torch.cat([inputs.input_ids, torch.tensor([tgt_ids], device=mo
del.device)], dim=1)

    with torch.no_grad():
        outputs = model(all_input_ids)
        logits = outputs.logits  # shape [1, seq_len, vocab_size]

    # target slice: last len(tgt_ids) tokens → logits that predicted them are previ
ous positions
    shift_logits = logits[:, -len(tgt_ids)-1:-1, :]  # shape [1, tgt_len, vocab]
    shift_labels = torch.tensor([tgt_ids], device=model.device)

    log_probs = torch.nn.functional.log_softmax(shift_logits, dim=-1)
    token_log_probs = log_probs.gather(2, shift_labels.unsqueeze(-1)).squeeze
(-1)  # [1, tgt_len]

    return token_log_probs.squeeze(0).cpu().numpy(), float(token_log_probs.su
m())

# ---------------------------
# Baseline: full prompt
# ---------------------------
```

```
baseline_token_logprobs, baseline_total_logprob = compute_logprobs(prompt,
target_output)
print(f"Baseline total log probability (full prompt): {baseline_total_logprob:.4f}
\n")

# ---------------------------
# Shapley approximation via permutation sampling
# ---------------------------
def prompt_from_subset(tokens_list, subset_indices):
    """Return string prompt containing tokens in original order for indices in sub
set_indices."""
    if not subset_indices:
        return ""  # we will handle empty subset specially
    return " ".join([tokens_list[i] for i in sorted(subset_indices)])

def approximate_shapley(tokens_list, target_text, model_logprob_func, num_p
ermutations=60, seed=0):
    """
    Approximate Shapley values via random permutation sampling.
    Returns dict mapping token index → shapley value (raw, on logprob scale).
    """
    random.seed(seed)
    n = len(tokens_list)
    phi = [0.0] * n
    cache = {}

    # Compute a safe v(empty). Try with empty prompt; if model fails, fallback t
o baseline (conservative).
    try:
        v_empty_arr, v_empty = model_logprob_func("", target_text)
    except Exception:
        # fallback: use minimal prompt as full prompt (conservative fallback)
        v_empty = baseline_total_logprob  # not ideal but avoids crashes
        print("Warning: model failed for empty prompt; using baseline total logpr
ob as v(empty) fallback.")
    else:
        # If succeeded, use computed value
        pass

    def v_of_subset(sorted_tuple):
        # sorted_tuple: tuple of token indices
        if sorted_tuple in cache:
            return cache[sorted_tuple]
        prompt_text = prompt_from_subset(tokens_list, sorted_tuple)
        try:
            _, val = model_logprob_func(prompt_text, target_text)
        except Exception as e:
            # If empty prompt errors, fallback to v_empty or baseline
            val = v_empty if 'v_empty' in locals() else baseline_total_logprob
        cache[sorted_tuple] = val
        return val

    # Sample permutations
    for _ in tqdm(range(num_permutations), desc="Shapley permutations"):
        perm = list(range(n))
        random.shuffle(perm)
        S = []
        for idx in perm:
            S_sorted = tuple(sorted(S))
            v_S = v_of_subset(S_sorted)
            S_with_i = tuple(sorted(S + [idx]))
            v_Si = v_of_subset(S_with_i)
            marginal = v_Si - v_S
            phi[idx] += marginal
            S.append(idx)

    # Average
    shap_vals = [x / num_permutations for x in phi]
    return {i: shap_vals[i] for i in range(n)}

# Run approximation (tune num_permutations as needed)
num_perm = 60  # increase for better precision, at cost of compute
```

```
shapley_raw = approximate_shapley(prompt_tokens, target_output, compute_l
ogprobs, num_permutations=num_perm, seed=42)

# ---------------------------
# Normalize & print results (0..1 scaling)
# ---------------------------
raw_vals = np.array([shapley_raw[i] for i in range(len(prompt_tokens))])
# Shapley values can be negative/positive; for display we map to 0..1 by shiftin
g
minv, maxv = raw_vals.min(), raw_vals.max()
if maxv - minv > 1e-12:
    normalized = (raw_vals - minv) / (maxv - minv)
else:
    normalized = np.zeros_like(raw_vals)

print("Token-wise Shapley (raw logprob contribution):")
for tok, val in zip(prompt_tokens, raw_vals):
    print(f"{tok:<15} {val: .6f}")
print("\nNormalized 0..1 (for heatmap display):")
for tok, val in zip(prompt_tokens, normalized):
    print(f"{tok:<15} {val:.3f}")

# ---------------------------
# Heatmap visualization (green = important)
# ---------------------------
plt.figure(figsize=(max(10, len(prompt_tokens)), 1.6))
sns.heatmap([normalized], annot=[prompt_tokens], fmt="", cmap="RdYlGn",
cbar=True, xticklabels=False, yticklabels=False)
plt.title(f"Prompt Token Shapley Importance (approx, {num_perm} perms) - O
SS-20B")
plt.show()
```

💡 **60 Permutations**. Time taken → 1.5 **Minutes** 👇



```
Baseline total log probability (full prompt): -738.0000

Shapley permutations: 100%|██████████| 60/60 [01:44<00:00,  1.74s/it]Token-wise Shapley (raw logprob contribution):
explain         -97.745833
deep            -48.725000
leaning         -44.291667
in               7.916667
50              -64.341667
words           -77.562500

Normalized 0..1 (for heatmap display):
explain         0.000
deep            0.464
leaning         0.506
in              1.000
50              0.316
words           0.191
```

Prompt Token Shapley Importance (approx, 60 perms) - OSS-20B
explain
deep
leaning
in
50
words
1.00
0.75
0.50
0.25
0.00

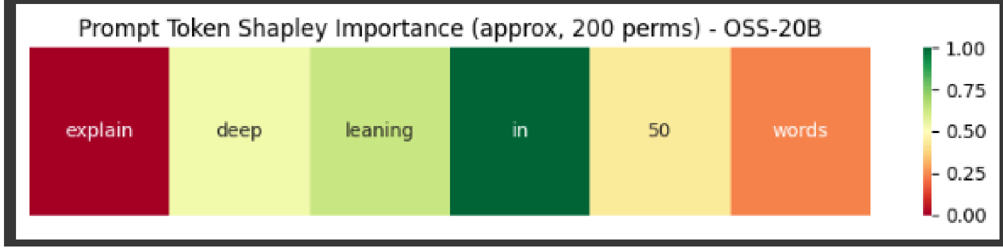
💡 **200 Permutations**. Time taken →4 **Minutes** 👇

```
# Shapley approx for prompt token importance using OSS-20B (Unsloth)
# Requires: unsloth model + tokenizer loaded as `model, tokenizer`
# If you haven't loaded, uncomment the load block and set model_name accordingly.

import torch
import numpy as np
from tqdm import tqdm
import random
import seaborn as sns
import matplotlib.pyplot as plt
```

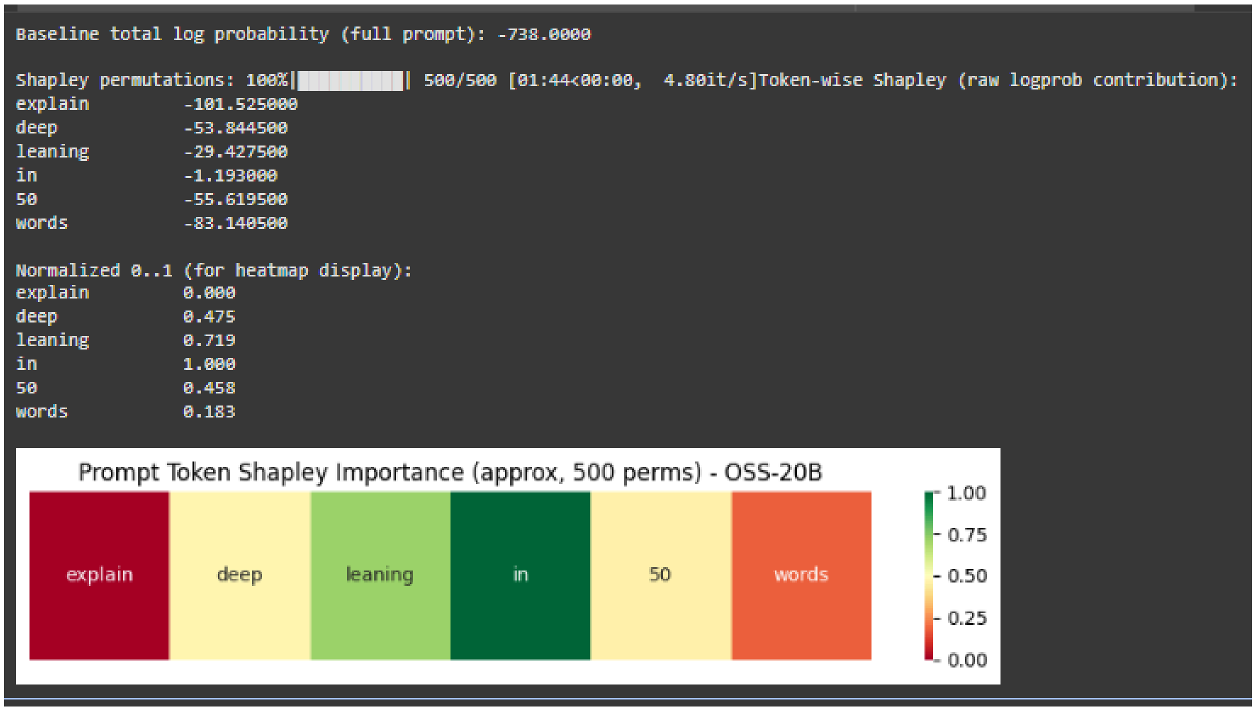
```
# ---------------------------
# Optional: load model/tokenizer (uncomment if not already loaded)
# ---------------------------
# from unsloth import FastLanguageModel
# model_name = "unsloth/gpt-oss-20b-unsloth-bnb-4bit"
# model, tokenizer = FastLanguageModel.from_pretrained(
#     model_name=model_name,
#     dtype=None,
#     max_seq_length=4096,
#     load_in_4bit=False,
#     full_finetuning=False
# )
# if torch.cuda.is_available():
#     model = model.to("cuda")
# model.eval()

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

# ---------------------------
# User prompt + target (kept same as your provided example)
# ---------------------------
prompt = "explain deep leaning in 50 words"
target_output = ("""
Deep learning is a subset of machine learning that uses multi-layered neural
networks to learn from vast amounts of data. Inspired by the human brain's st
ructure, these
networks automatically identify patterns and features, enabling them to perfor
m complex tasks
like image recognition, natural language processing, and autonomous driving
""")

# Work at whitespace token granularity to stay consistent with your previous c
ode
prompt_tokens = prompt.split()

# ---------------------------
# Reuse compute_logprobs function (returns per-token logprobs and total)
# ---------------------------
def compute_logprobs(prompt_text, target_text):
    """
    Returns (token_logprob_array, total_logprob)
    - token_logprob_array: numpy array of shape (tgt_len,) with log-probs for e
ach target token
    - total_logprob: float sum of token log-probs
    """
    # Build model-specific input using apply_chat_template (same pattern as yo
ur code)
    inputs = tokenizer.apply_chat_template(
        [{"role": "user", "content": prompt_text}],
        add_generation_prompt=True,
        return_tensors="pt",
        return_dict=True
    ).to(model.device)

    tgt_ids = tokenizer(target_text).input_ids
    # cat into a single tensor
    all_input_ids = torch.cat([inputs.input_ids, torch.tensor([tgt_ids], device=mo
del.device)], dim=1)

    with torch.no_grad():
        outputs = model(all_input_ids)
        logits = outputs.logits  # shape [1, seq_len, vocab_size]

    # target slice: last len(tgt_ids) tokens → logits that predicted them are previ
ous positions
    shift_logits = logits[:, -len(tgt_ids)-1:-1, :]  # shape [1, tgt_len, vocab]
    shift_labels = torch.tensor([tgt_ids], device=model.device)

    log_probs = torch.nn.functional.log_softmax(shift_logits, dim=-1)
    token_log_probs = log_probs.gather(2, shift_labels.unsqueeze(-1)).squeeze
(-1)  # [1, tgt_len]
```

```
    return token_log_probs.squeeze(0).cpu().numpy(), float(token_log_probs.su
m())

# ---------------------------
# Baseline: full prompt
# ---------------------------
baseline_token_logprobs, baseline_total_logprob = compute_logprobs(prompt,
target_output)
print(f"Baseline total log probability (full prompt): {baseline_total_logprob:.4f}
\n")

# ---------------------------
# Shapley approximation via permutation sampling
# ---------------------------
def prompt_from_subset(tokens_list, subset_indices):
    """Return string prompt containing tokens in original order for indices in sub
set_indices."""
    if not subset_indices:
        return ""  # we will handle empty subset specially
    return " ".join([tokens_list[i] for i in sorted(subset_indices)])

def approximate_shapley(tokens_list, target_text, model_logprob_func, num_p
ermutations=200, seed=0):
    """
    Approximate Shapley values via random permutation sampling.
    Returns dict mapping token index → shapley value (raw, on logprob scale).
    """
    random.seed(seed)
    n = len(tokens_list)
    phi = [0.0] * n
    cache = {}

    # Compute a safe v(empty). Try with empty prompt; if model fails, fallback t
o baseline (conservative).
    try:
        v_empty_arr, v_empty = model_logprob_func("", target_text)
    except Exception:
        # fallback: use minimal prompt as full prompt (conservative fallback)
        v_empty = baseline_total_logprob  # not ideal but avoids crashes
        print("Warning: model failed for empty prompt; using baseline total logpr
ob as v(empty) fallback.")
    else:
        # If succeeded, use computed value
        pass

    def v_of_subset(sorted_tuple):
        # sorted_tuple: tuple of token indices
        if sorted_tuple in cache:
            return cache[sorted_tuple]
        prompt_text = prompt_from_subset(tokens_list, sorted_tuple)
        try:
            _, val = model_logprob_func(prompt_text, target_text)
        except Exception as e:
            # If empty prompt errors, fallback to v_empty or baseline
            val = v_empty if 'v_empty' in locals() else baseline_total_logprob
        cache[sorted_tuple] = val
        return val

    # Sample permutations
    for _ in tqdm(range(num_permutations), desc="Shapley permutations"):
        perm = list(range(n))
        random.shuffle(perm)
        S = []
        for idx in perm:
            S_sorted = tuple(sorted(S))
            v_S = v_of_subset(S_sorted)
            S_with_i = tuple(sorted(S + [idx]))
            v_Si = v_of_subset(S_with_i)
            marginal = v_Si - v_S
            phi[idx] += marginal
            S.append(idx)
    # Average
    shap_vals = [x / num_permutations for x in phi]
    return {i: shap_vals[i] for i in range(n)}

# Run approximation (tune num_permutations as needed)
num_perm = 200  # increase for better precision, at cost of compute
shapley_raw = approximate_shapley(prompt_tokens, target_output, compute_l
ogprobs, num_permutations=num_perm, seed=42)

# ---------------------------
# Normalize & print results (0..1 scaling)
# ---------------------------
raw_vals = np.array([shapley_raw[i] for i in range(len(prompt_tokens))])
# Shapley values can be negative/positive; for display we map to 0..1 by shiftin
g
minv, maxv = raw_vals.min(), raw_vals.max()
if maxv - minv > 1e-12:
    normalized = (raw_vals - minv) / (maxv - minv)
else:
    normalized = np.zeros_like(raw_vals)

print("Token-wise Shapley (raw logprob contribution):")
for tok, val in zip(prompt_tokens, raw_vals):
    print(f"{tok:<15} {val: .6f}")
print("\nNormalized 0..1 (for heatmap display):")
for tok, val in zip(prompt_tokens, normalized):
    print(f"{tok:<15} {val:.3f}")

# ---------------------------
# Heatmap visualization (green = important)
# ---------------------------
plt.figure(figsize=(max(10, len(prompt_tokens)), 1.6))
sns.heatmap([normalized], annot=[prompt_tokens], fmt="", cmap="RdYlGn",
cbar=True, xticklabels=False, yticklabels=False)
plt.title(f"Prompt Token Shapley Importance (approx, {num_perm} perms) - O
SS-20B")
plt.show()
```

```
Baseline total log probability (full prompt): -738.0000

Shapley permutations: 100%|██████████| 200/200 [01:45<00:00,  1.90it/s]Token-wise Shapley (raw logprob contribution):
explain         -103.967500
deep            -47.955000
leaning         -37.767500
in               1.307500
50              -57.595000
words           -78.772500

Normalized 0..1 (for heatmap display):
explain         0.000
deep            0.532
leaning         0.629
in              1.000
50              0.440
words           0.239
```



Permutations= 500:



```
Baseline total log probability (full prompt): -738.0000

Shapley permutations: 100%|██████████| 500/500 [01:44<00:00,  4.80it/s]Token-wise Shapley (raw logprob contribution):
explain         -101.525000
deep            -53.844500
leaning         -29.427500
in              -1.193000
50              -55.619500
words           -83.140500

Normalized 0..1 (for heatmap display):
explain         0.000
deep            0.475
leaning         0.719
in              1.000
50              0.458
words           0.183
```

Prompt Token Shapley Importance (approx, 500 perms) - OSS-20B
explain
deep
leaning
in
50
words
1.00
0.75
0.50
0.25
0.00

Generate response

- The following code generates 2 responses: with and without thinking

```
from transformers import TextStreamer

messages = [
    {"role": "user", "content": "who threw apple at gregor samsa"},
]

# Function to generate and process output
def generate_and_process(messages, model, tokenizer, reasoning_effort="lo
w"):
    inputs = tokenizer.apply_chat_template(
        messages,
        add_generation_prompt = True,
        return_tensors = "pt",
        return_dict = True,
        reasoning_effort = reasoning_effort, # Set reasoning effort
    ).to("cuda")
    # Generate without streamer to get the full output at once
    outputs = model.generate(**inputs, max_new_tokens = 512) # Removed str
eamer

    # Decode the output
    generated_text = tokenizer.decode(outputs[0], skip_special_tokens=False)
# Keep special tokens to identify channels

    # --- Parse the output to separate channels ---
    final_response = ""
    analysis = ""
    commentary = ""

    # Split by channel markers
    parts = generated_text.split("<|channel|>")
    if len(parts) > 1:
        for part in parts[1:]:
            if "<|message|>" in part:
                channel_name, message_content = part.split("<|message|>", 1)
                message_content = message_content.split("<|end|>", 1)[0].strip()
                if channel_name.strip() == "final":
                    final_response += message_content + "\n"
                elif channel_name.strip() == "analysis":
                    analysis += message_content + "\n"
                elif channel_name.strip() == "commentary":
                    commentary += message_content + "\n"
    else:
        # If no channel markers found, assume the whole output is the final respo
nse
        final_response = generated_text

    # Combine analysis and commentary into 'thinking'
    thinking_output = ""
    if analysis:
        thinking_output += "Analysis:\n" + analysis
    if commentary:
        if thinking_output:
            thinking_output += "\n"
        thinking_output += "Commentary:\n" + commentary

    # Remove <|return|> token from final response
    final_response = final_response.replace("<|return|>", "").strip()

    return thinking_output, final_response

# --- Example with thinking (low reasoning effort) ---
print("--- With Thinking (low reasoning effort) ---")
thinking, final = generate_and_process(messages, model, tokenizer, reasonin
g_effort="low")
print(thinking)
print("\nFinal Response:")
print(final)

# --- Example without thinking (reasoning_effort="none") ---
print("\n--- Without Thinking (reasoning_effort=\"none\") ---")
thinking_no_thinking, final_no_thinking = generate_and_process(messages, m
odel, tokenizer, reasoning_effort="none")
print(thinking_no_thinking)
print("\nFinal Response:")
print(final_no_thinking)
```

```
Output:

--- With Thinking (low reasoning effort) ---
Analysis:
We have a literary question: "who threw apple at Gregor Samsa". In Kafka's "T
he Metamorphosis", there's a scene where someone throws a fruit at Gregor.
The protagonist's brother? It's likely that the father throws an apple? I recall a
scene: Two guys, a boy named Benda? It's the neighbor. Actually, in the story,
Mr. and Mrs. Samsa throw apples at each other? Wait: In chapter II, a boy nam
```

ed Benda throws apples at Hans and Jutta? Actually, the detail: In the story aft
er the transformation, the family finds a glass of wine, then a neighbor, a boy, i
s thrown a glass of wine? But the apple throw is by the girl (the sister?) Actuall
y, the "apple thrower" is the child of the man the Samsas go to for the apartm
ent. It's unclear.

Wait: In Kafka's The Metamorphosis, there is a scene where the family receive
s an apple from the "young woman" of the house; he is thrown an apple?

But the question: "who threw apple at Gregor Samsa". I think it's the family of
the "München"? Actually, I recall reading that the father inadvertently throws a
n apple. So answer: The father? Or the sister? Perhaps it's the character "Mr.
Jupp"? But I'm not sure.

Let's search memory: In chapter two, a boy called Tom throws an apple at the
brother? Eh.

Wait, I recall that the old lady, named "Gretchen", with the "apple thrown at Gr
egor" is the boy "Graham". But it's not correct.

I think answer: The brother, Frederic, throws an apple. But I don't recall.

I'll give a general answer: The apple is thrown by the family.

Let's answer generically.

Final Response:
In Franz Kafka's *The Metamorphosis*, the only "apple-throw" incident that's o
ften cited is when the young boy who visits the Samsa household—often refer
red to as the **little boy** or **boy-neighbor**—throws a fruit at the family. I
t's not the protagonist himself, but a neighbor's son, sometimes called **Hans
J.**, who, in a fit of excitement, hurls an apple toward the family's home. So, t
o answer your question: it was the **young boy ("the boy-neighbor")** who th
rew an apple at Gregor Samsa.

```
--- Without Thinking (reasoning_effort="none") ---
Analysis:
User asks: "who threw apple at gregor samsa". They refer to Kafka's The Met
amorphosis where Gregor Samsa is in an apartment with a door and he says
"My hand shook". The apple? I recall it's about a story of someone throwing a
n apple into his house? Wait: In The Metamorphosis, a key scene is that befor
e the transformation, we have his wife and mother. But apple: maybe referring
to an apple thrown by a neighbor? Not sure. Search? Probably the answer: "W
e don't know". The answer: the apple was thrown by a waiter? In an anecdote:
In the story of the "Ain't this an apple?" Actually the apple thrown at Gregor by
a passerby? Let's respond that it's unclear.



Final Response:
In **Kafka’s _The Metamorphosis_** there is no episode where an apple is liter
ally thrown at Gregor Samsa, and no character is explicitly identified as the thr
ower. The story tells of a strange transformation of a young man into a giant in
sect and the reactions of his family and neighbors—the only “thing” that come
s to mind is **the apple-throwing prank that the narrator describes in the earli
er “The Apple Tree” story** in which the protagonist is thrown a fruit by a pas
ser-by.

Because the question appears to mix details from different works (the novel
“The Metamorphosis” and the short story “The Apple Tree”) it is impossible to
answer it definitively. The only character known to be involved in a “throwing”
event at all is the **protagonist’s father** in the short story in the “The apple-t
ree”, who throws an apple (to the floor) to the protagonist in order to “spice th
e narrative and give it a taste of humanity".

In short: there is no clear answer in _The Metamorphosis_, because no such e
vent occurs there.
```

What 4-bit quantization is and how it affects log-probs and Shapley values

A. What is 4-bit quantization

- Neural network weights are normally stored in float32 or float16 (32 or 16 bits per number).
- **4-bit quantization** stores weights at much lower precision (4 bits per number) using special schemes (e.g., per-channel scales + integer storage) so the model uses far less memory.
- Popular inference libraries (bitsandbytes, custom MXFP4, etc.) implement quantization plus scaling factors to preserve information.

B. Why people use it

- Allows models with billions of parameters (20B, 70B, 120B) to run on GPUs with limited memory (e.g., 16–48 GB).
- Dramatically reduces RAM and VRAM requirements and download size.

🧠 What is **Unsloth**?

Unsloth is an open-source library designed **to make large models** (like 7B–120B parameter LLMs) **faster, smaller, and easier to run locally** — especially on limited GPUs like Colab's T4 or your own 16–24 GB GPU.

It's basically a *speed and memory optimization layer* over **Hugging Face Transformers**.

⚙️ What is **FastLanguageModel**?

It's Unsloth's version of Hugging Face's model loader.

It automatically:

- Detects hardware (GPU/CPU)
- Loads models with quantization (reduces memory)
- Optimizes runtime (uses bitsandbytes or Triton kernels)
- Keeps the same interface (you can use `.generate()` or compute logits/log-probs)

You can still get **log probabilities**, gradients, etc. — it's just more efficient.

🔩 What is **torch** doing here?

`torch` = **PyTorch**, the deep learning engine underneath.

It handles:

- GPU computation (tensor math)
- Model layers and weights
- Logit → log-probability conversion (via `torch.nn.functional.log_softmax`)
- Automatic batching, precision, etc.

Basically, **PyTorch = the engine**, **Unsloth = the optimizer**, **Transformers = the interface**.

🧩 Diagram: How Log Probabilities Work

```
Prompt → [Tokenizer] → Input IDs (numbers)
           ↓
       Model (Transformer layers)
           ↓
       Logits  (raw scores for every possible next token)
           ↓
   log_softmax(logits)
           ↓
   Log probabilities (confidence for each token)
           ↓
   Compare across prompts → importance / SHAP scores
```

- **Prompt**: your question or instruction.
- **Tokenizer**: splits text into tokens (like “explain”, “deep”, “learning”).
- **Model (OSS-20B)**: predicts *how likely* each possible next token is.
- **Logits**: big matrix of scores, one per vocab token.
- **log softmax**: converts those scores to log probabilities (so they sum to 1 in probability space).

- **Log probabilities** tell you: “how confident the model is in producing this token.”
- **SHAP or Leave-One-Out** then checks: *if I remove a token from the prompt, how much does the total log probability of the correct answer drop?* → that’s the token’s importance.

How to compare Log Probabilities?

- Compute `logprobs` for each token in the output for both prompts.
- Calculate **difference** (Δ logprob):

  ```
  delta_logprob = logprob_promptA - logprob_promptB
  ```

- A **larger drop** in log probability when a prompt word is removed or changed → that prompt word was **important** for generating that output.

Think of it like:

> “How much less confident the model becomes if I remove this part of the prompt?”

Heat Map

```
# Requires: unsloth FastLanguageModel (model, tokenizer) already loaded.
# If not loaded, uncomment the "Load model" block and run (be mindful of VR
AM / quantization options).

import torch
import numpy as np
from tqdm import tqdm
import random
import math
import seaborn as sns
```

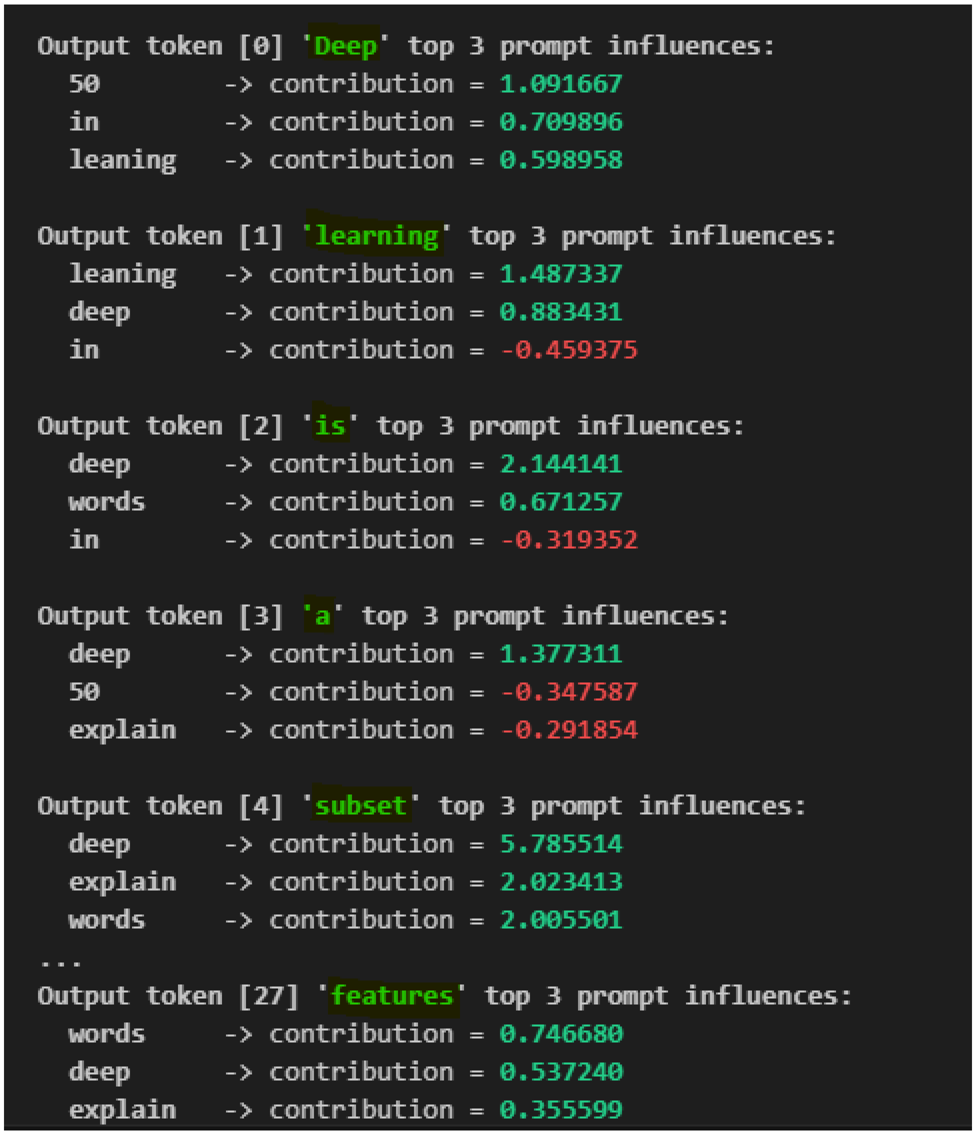
```
import matplotlib.pyplot as plt
from typing import List, Tuple

# ---------------------------
# Optional: load model/tokenizer (uncomment if not already loaded)
# ---------------------------
# from unsloth import FastLanguageModel
# model_name = "unsloth/gpt-oss-20b-unsloth-bnb-4bit"
# model, tokenizer = FastLanguageModel.from_pretrained(
#     model_name=model_name,
#     dtype=None,
#     max_seq_length=4096,
#     load_in_4bit=False,   # model already pre-quantized
#     full_finetuning=False,
# )
# if torch.cuda.is_available():
#     model = model.to("cuda")
# model.eval()

device = model.device if hasattr(model, "device") else ("cuda" if torch.cuda.is
_available() else "cpu")

# ---------------------------
# User inputs (change these)
# ---------------------------
prompt = "explain deep leaning in 50 words"
# A longer multi-sentence output to test big outputs:
target_output = (
    "Deep learning is a subset of machine learning that uses multi-layer neural
networks "
    "to learn patterns from very large datasets. These networks can automatica
lly extract "
    "features from raw inputs, enabling tasks like image recognition, speech pr
ocessing, and "
    "language understanding. Modern architectures scale to billions of paramet
ers and require "
```

```
    "careful regularization and optimization to perform well."
)

# Use "words" for prompt tokens (human-friendly). Internally we pass strings
to tokenizer.
prompt_words = prompt.split()

# Shapley settings
num_permutations = 60   # start small (40-100). Increase for more accurate S
hapley.
random_seed = 42

# Optional: limit output length (tokens) for speed; set None to use full target
max_output_tokens = None  # e.g., 120 or None

# ---------------------------
# Helper: compute per-target-token log probs for a given prompt text
# Returns: numpy array shape (T,) of logprobs (natural log), and float total log
prob
# ---------------------------
def compute_logprobs_for_prompt(prompt_text: str, target_text: str) → Tuple[n
p.ndarray, float]:
    """
    Runs model on [prompt_text + target_text] and returns per-target-token log
probs (numpy array)
    and the total logprob (sum).
    """
    # Build input in the same way as used earlier (chat template)
    inputs = tokenizer.apply_chat_template(
        [{"role": "user", "content": prompt_text}],
        add_generation_prompt=True,
        return_tensors="pt",
        return_dict=True
    ).to(device)

    tgt_ids = tokenizer(target_text).input_ids
    if max_output_tokens is not None and len(tgt_ids) > max_output_tokens:
        tgt_ids = tgt_ids[:max_output_tokens]

    # Stack into single tensor [1, seq_len]
    all_input_ids = torch.cat([inputs.input_ids, torch.tensor([tgt_ids], device=de
vice)], dim=1)

    with torch.no_grad():
        outputs = model(all_input_ids)          # model forward
        logits = outputs.logits                 # shape [1, seq_len, vocab_size]

    # Align logits that predicted target tokens
    T = len(tgt_ids)
    shift_logits = logits[:, -T-1:-1, :]       # shape [1, T, vocab_size]
    labels = torch.tensor([tgt_ids], device=device)

    log_probs = torch.nn.functional.log_softmax(shift_logits, dim=-1)  # [1,T,V]
    token_log_probs = log_probs.gather(2, labels.unsqueeze(-1)).squeeze(-1)  #
[1,T]

    arr = token_log_probs.squeeze(0).cpu().numpy()   # numpy array length T
    total = float(arr.sum())
    return arr, total

# Quick baseline check
print("Computing baseline (full prompt) logprobs... (this will run the model onc
e)")
baseline_per_token_logprobs, baseline_total = compute_logprobs_for_prompt
(prompt, target_output)
T = len(baseline_per_token_logprobs)
print(f"Baseline total logprob: {baseline_total:.4f} over {T} output tokens")

# ---------------------------
# Shapley approx for each output token:
# We will compute shapley contributions per prompt word i for each output to
ken j.
```

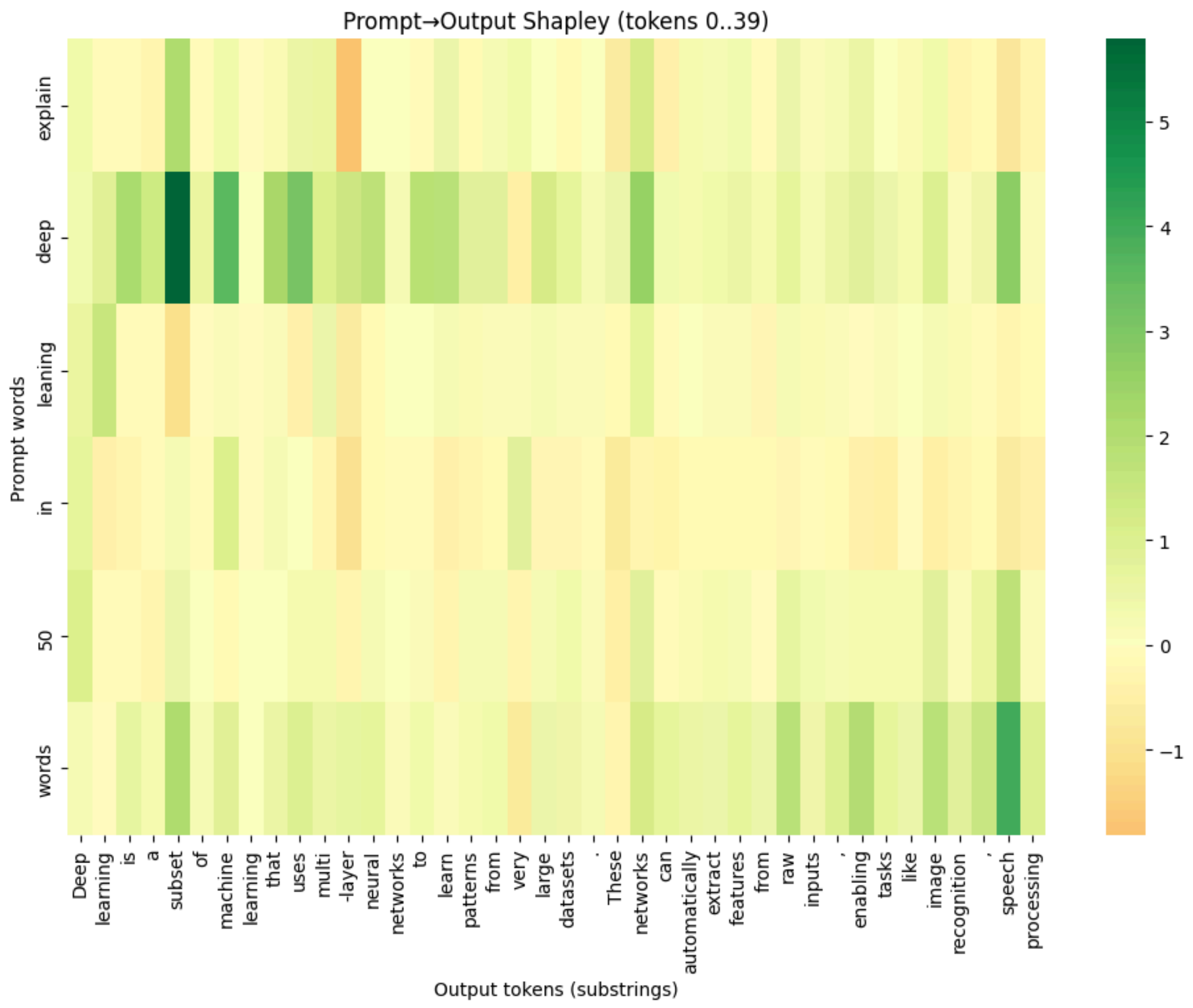
```
# Algorithm:
#  - sample permutations; for each permutation, walk adding tokens; compute
v(S) for each S
#  - v(S) is the per-token logprob vector; marginal contribution for token i is v
(S ∪ {i}) - v(S) (vector)
#  - accumulate contributions per (i, j)
# Caching: store v(S) in dict for reuse
# ---------------------------
def prompt_from_indices(words: List[str], indices_tuple: Tuple[int, ...]) → str:
    if not indices_tuple:
        return ""  # empty prompt
    return " ".join([words[i] for i in sorted(indices_tuple)])

def approximate_shapley_prompt_to_output(words: List[str],
                                         target_text: str,
                                         model_fn,
                                         num_permutations: int = 60,
                                         seed: int = 0):
    random.seed(seed)
    n = len(words)
    # We'll accumulate raw contributions (logprob units) in a matrix (n x T)
    # Initialize after we know T
    # Caching: map subset_tuple → per-token logprobs (numpy array)
    cache = {}

    # try v(empty)
    try:
        empty_arr, empty_total = model_fn("", target_text)
    except Exception as e:
        # fallback: use baseline_total (conservative) for empty subsets (documen
t this)
        print("Warning: model failed on empty prompt. Using baseline_total as fal
lback for v(empty).")
        empty_arr = np.zeros_like(baseline_per_token_logprobs)
        empty_total = baseline_total
    # initialize accumulator
    accumulator = np.zeros((n, len(baseline_per_token_logprobs)), dtype=float)

    def v_of_subset(sorted_tuple):
        if sorted_tuple in cache:
            return cache[sorted_tuple]
        prompt_text = prompt_from_indices(words, sorted_tuple)
        try:
            arr, _ = model_fn(prompt_text, target_text)
        except Exception:
            # fallback
            arr = empty_arr
        cache[sorted_tuple] = arr
        return arr

    print(f"Running {num_permutations} permutations (this may be slow).")
    for _ in tqdm(range(num_permutations), desc="Shapley perms"):
        perm = list(range(n))
        random.shuffle(perm)
        S = []
        for idx in perm:
            S_sorted = tuple(sorted(S))
            S_with = tuple(sorted(S + [idx]))

            vS = v_of_subset(S_sorted)
            vSi = v_of_subset(S_with)

            marginal = vSi - vS   # vector length T
            accumulator[idx] += marginal
            S.append(idx)

    shap_raw = accumulator / float(num_permutations)   # shape (n, T)
    return shap_raw, cache

# Run approximation (this is the heavy step)
shap_raw_matrix, cache_used = approximate_shapley_prompt_to_output(
    prompt_words, target_output, compute_logprobs_for_prompt,
    num_permutations=num_permutations, seed=random_seed
)

# ----------------------------
# Postprocess: make readable labels for output tokens
# ----------------------------
# Get token-level strings for target tokens (decoded)
tgt_ids_full = tokenizer(target_output).input_ids
if max_output_tokens is not None and len(tgt_ids_full) > max_output_tokens:
    tgt_ids = tgt_ids_full[:max_output_tokens]
else:
    tgt_ids = tgt_ids_full

output_token_strs = [tokenizer.decode([tid]).strip() for tid in tgt_ids]  # list leng
th T
prompt_labels = prompt_words

# ----------------------------
# Visualization helpers
# ----------------------------
def plot_chunked_heatmaps(shap_matrix, prompt_labels, output_labels, chunk
_size=40, cmap="RdYlGn"):
    """
    Make multiple heatmaps for chunks of output tokens.
    shap_matrix: (n, T)
    """
    n, T = shap_matrix.shape
    for start in range(0, T, chunk_size):
        end = min(T, start + chunk_size)
        chunk = shap_matrix[:, start:end]
        out_labels_chunk = output_labels[start:end]
        plt.figure(figsize=(max(10, len(prompt_labels)*0.25), max(2, (end-start)*
0.12 + 3)))
        sns.heatmap(chunk, xticklabels=out_labels_chunk, yticklabels=prompt_la
bels, cmap=cmap, center=0.0)
```

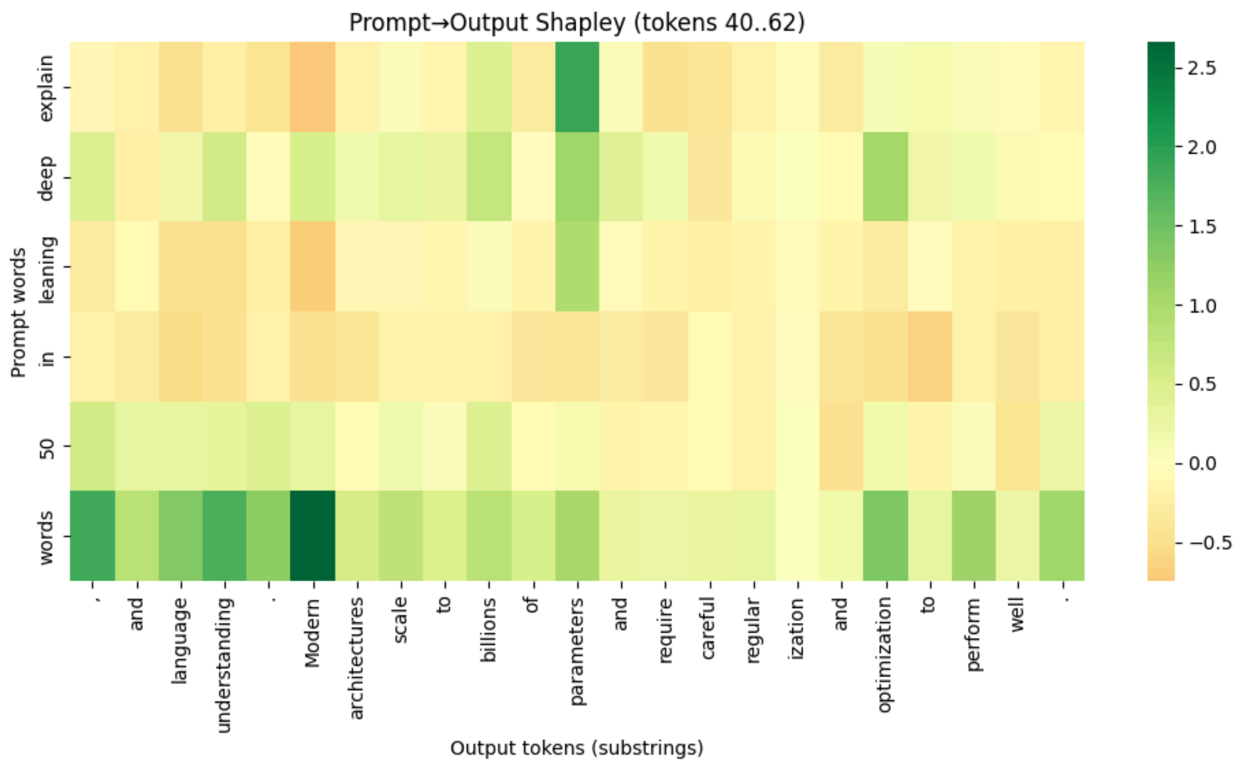
```
        plt.xticks(rotation=90)
        plt.title(f"Prompt→Output Shapley (tokens {start}..{end-1})")
        plt.xlabel("Output tokens (substrings)")
        plt.ylabel("Prompt words")
        plt.tight_layout()
        plt.show()

def aggregate_by_sentences(shap_matrix, output_text: str, prompt_labels=No
ne, agg="sum_abs"):
    """
    Aggregate contributions for tokens into sentence-level contributions.
    agg="sum_abs" (sum of absolute contributions) or "sum" (signed sum), or
"mean_abs".
    Returns (n_words, n_sentences) matrix and sentence strings.
    """
    # naive sentence split (split on period). For production use, use nltk.sent_to
kenize
    sents = [s.strip() for s in output_text.replace("\n", " ").split(".") if s.strip()]
    sent_token_ranges = []
    pos = 0
    tgt_tokens = [tokenizer.decode([tid]).strip() for tid in tgt_ids]
    token_texts = " ".join(tgt_tokens)
    # We'll map tokens to sentence by approximate substring matching (simple
approach)
    # Simpler: re-tokenize each sentence and count tokens
    token_counts = []
    for sent in sents:
        toks = tokenizer(sent).input_ids
        token_counts.append(len(toks))
    idx = 0
    ranges = []
    for c in token_counts:
        ranges.append((idx, idx + c))
        idx += c

    # Build aggregated matrix
    n_words = shap_matrix.shape[0]
    n_sents = len(ranges)
    agg_mat = np.zeros((n_words, n_sents))
    for si, (a, b) in enumerate(ranges):
        if agg == "sum_abs":
            agg_mat[:, si] = np.abs(shap_matrix[:, a:b]).sum(axis=1)
        elif agg == "sum":
            agg_mat[:, si] = shap_matrix[:, a:b].sum(axis=1)
        elif agg == "mean_abs":
            agg_mat[:, si] = np.abs(shap_matrix[:, a:b]).mean(axis=1)
        else:
            agg_mat[:, si] = np.abs(shap_matrix[:, a:b]).sum(axis=1)
    return agg_mat, sents

# ---------------------------
# Plot chunked token-level heatmaps (use smaller chunk_size if T is large)
# ---------------------------
chunk_size = 40
plot_chunked_heatmaps(shap_raw_matrix, prompt_labels, output_token_strs,
chunk_size=chunk_size)

# ---------------------------
# Plot aggregated by sentences
# ---------------------------
agg_mat, sents = aggregate_by_sentences(shap_raw_matrix, target_output, pr
ompt_labels, agg="sum_abs")
plt.figure(figsize=(10, max(3, len(prompt_labels)*0.25)))
sns.heatmap(agg_mat, xticklabels=[ (s[:60] + '...') if len(s)>60 else s for s in s
ents ], yticklabels=prompt_labels, cmap="RdYlGn")
plt.xticks(rotation=45, ha="right")
plt.title("Prompt → Sentence aggregated Shapley (sum abs contributions)")
plt.xlabel("Output sentences (truncated)")
plt.ylabel("Prompt words")
plt.tight_layout()
plt.show()
```

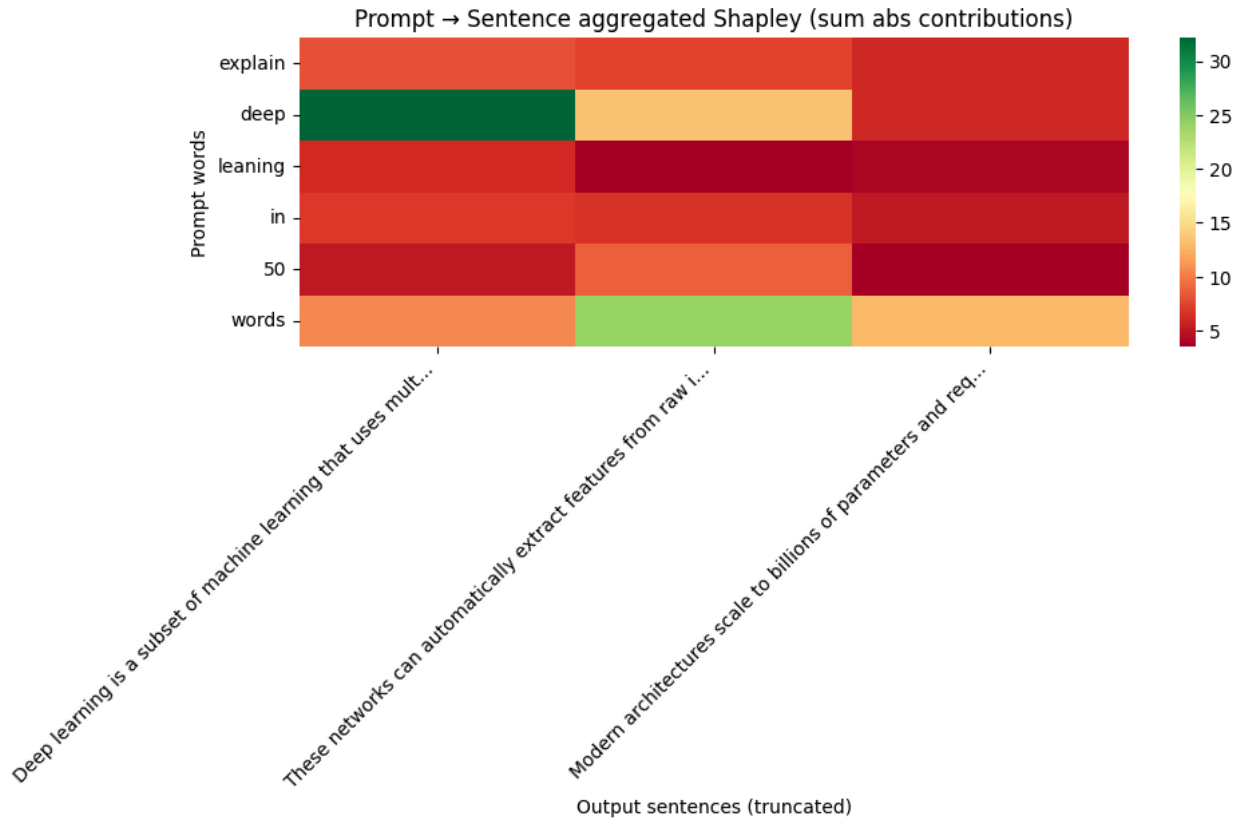
```
# ---------------------------
# Top-k prompt words per output token (for first N tokens) - useful when T lar
ge
# ---------------------------
def topk_prompt_for_each_output(shap_matrix, prompt_labels, output_labels, t
opk=3, limit_tokens=60):
    n, T = shap_matrix.shape
    for j in range(min(T, limit_tokens)):
        col = shap_matrix[:, j]
        # show absolute magnitude ranking (in logprob units)
        idxs = np.argsort(-np.abs(col))[:topk]
        print(f"Output token [{j}] '{output_labels[j]}' top {topk} prompt influence
s:")
        for rank, i in enumerate(idxs, 1):
            print(f"  {prompt_labels[i]} → contribution = {col[i]: .6f}")
    print("...done (showed first", min(T, limit_tokens), "output tokens).")

topk_prompt_for_each_output(shap_raw_matrix, prompt_labels, output_token_
strs, topk=3, limit_tokens=40)
```



```
Output token [0] 'Deep' top 3 prompt influences:
  50         -> contribution = 1.091667
  in         -> contribution = 0.709896
  leaning    -> contribution = 0.598958

Output token [1] 'learning' top 3 prompt influences:
  leaning    -> contribution = 1.487337
  deep       -> contribution = 0.883431
  in         -> contribution = -0.459375

Output token [2] 'is' top 3 prompt influences:
  deep       -> contribution = 2.144141
  words      -> contribution = 0.671257
  in         -> contribution = -0.319352

Output token [3] 'a' top 3 prompt influences:
  deep       -> contribution = 1.377311
  50         -> contribution = -0.347587
  explain    -> contribution = -0.291854

Output token [4] 'subset' top 3 prompt influences:
  deep       -> contribution = 5.785514
  explain    -> contribution = 2.023413
  words      -> contribution = 2.005501
...
Output token [27] 'features' top 3 prompt influences:
  words      -> contribution = 0.746680
  deep       -> contribution = 0.537240
  explain    -> contribution = 0.355599
```


Prompt→Output Shapley (tokens 0..39)
Prompt words
explain
deep
leaning
in
50
words
Output tokens (substrings)
Deep
learning
is
a
subset
of
machine
learning
that
uses
multi
-layer
neural
networks
to
learn
patterns
from
very
large
datasets
.
These
networks
can
automatically
extract
features
from
raw
inputs
,
enabling
tasks
like
image
recognition
,
speech
processing
5
4
3
2
1
0
−1

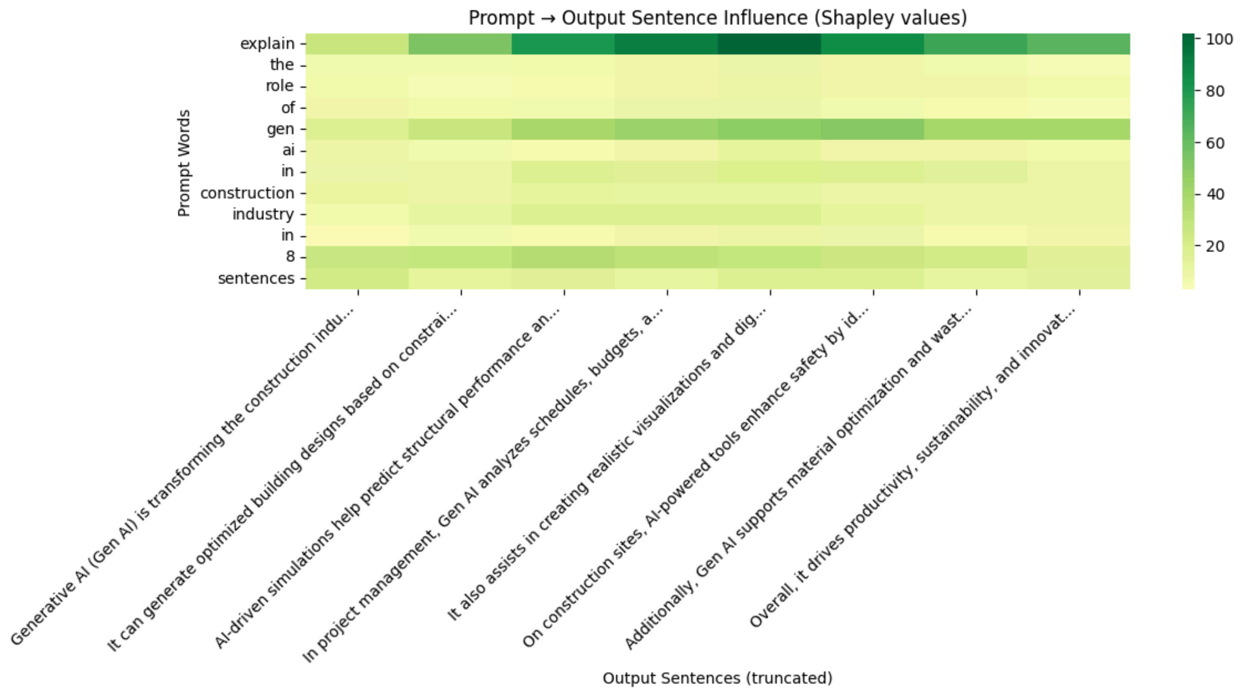

Prompt→Output Shapley (tokens 40..62)
Prompt words
explain
deep
leaning
in
50
words
,
and
language
understanding
.
Modern
architectures
scale
to
billions
of
parameters
and
require
careful
regular
ization
and
optimization
to
perform
well
.
Output tokens (substrings)
2.5
2.0
1.5
1.0
0.5
0.0
−0.5


Prompt → Sentence aggregated Shapley (sum abs contributions)
Prompt words
explain
deep
leaning
in
50
words
Deep learning is a subset of machine learning that uses mult...
These networks can automatically extract features from raw i...
Modern architectures scale to billions of parameters and req...
Output sentences (truncated)
30
25
20
15
10
5

Impact of single word on sentences

```
# ==============================
# SHAPLEY VISUALIZATION (FAST DEMO)
# ==============================
import torch
import numpy as np
import random
from tqdm import tqdm
import seaborn as sns
import matplotlib.pyplot as plt
from unsloth import FastLanguageModel
from transformers import TextStreamer

# ---------------------------
# STEP 1: Load Unsloth Model (lightweight demo)
# ---------------------------
# model_name = "unsloth/gpt-oss-20b-unsloth-bnb-4bit"
# model, tokenizer = FastLanguageModel.from_pretrained(
#     model_name=model_name,
#     dtype=None,
#     max_seq_length=4096,
#     load_in_4bit=False,
#     full_finetuning=False,
# )
device = "cuda" if torch.cuda.is_available() else "cpu"
#model = model.to(device)
model.eval()

# ---------------------------
# STEP 2: Prompt & hardcoded model output
# ---------------------------
prompt = (
    "explain the role of gen ai in construction industry in 8 sentences"
)
prompt_words = prompt.split()

# Hardcoded output (8–10 sentences)
output_text = (
    "Generative AI (Gen AI) is transforming the construction industry by automa
ting design, planning, and project management tasks. It can generate optimize
d building designs based on constraints like cost, materials, and sustainability
goals. AI-driven simulations help predict structural performance and detect po
tential issues before construction begins, reducing errors and rework. In proje
ct management, Gen AI analyzes schedules, budgets, and workforce data to p
redict delays and suggest corrective actions. It also assists in creating realisti
c visualizations and digital twins, improving communication between architect
s, engineers, and clients. On construction sites, AI-powered tools enhance saf
ety by identifying hazards through real-time image and sensor analysis. Additi
onally, Gen AI supports material optimization and waste reduction by recomm
ending efficient resource usage. Overall, it drives productivity, sustainability, a
nd innovation across the entire construction lifecycle."
)

# ---------------------------
# STEP 3: Compute per-token logprobs
# ---------------------------
def compute_logprobs_for_prompt(prompt_text: str, target_text: str):
    inputs = tokenizer.apply_chat_template(
        [{"role": "user", "content": prompt_text}],
        add_generation_prompt=True,
        return_tensors="pt",
        return_dict=True
    ).to(device)

    tgt_ids = tokenizer(target_text).input_ids
    all_input_ids = torch.cat([inputs.input_ids, torch.tensor([tgt_ids], device=de
vice)], dim=1)
```

```
    with torch.no_grad():
        outputs = model(all_input_ids)
        logits = outputs.logits

    T = len(tgt_ids)
    shift_logits = logits[:, -T-1:-1, :]
    labels = torch.tensor([tgt_ids], device=device)

    log_probs = torch.nn.functional.log_softmax(shift_logits, dim=-1)
    token_log_probs = log_probs.gather(2, labels.unsqueeze(-1)).squeeze(-1)
    arr = token_log_probs.squeeze(0).cpu().numpy()
    return arr, float(arr.sum())

# ---------------------------
# STEP 4: Shapley approximation (fast)
# ---------------------------
def prompt_from_indices(words, indices_tuple):
    if not indices_tuple:
        return ""
    return " ".join([words[i] for i in sorted(indices_tuple)])

def approximate_shapley_sentence(words, target_text, model_fn, num_permut
ations=10, seed=0):
    random.seed(seed)
    n = len(words)
    cache = {}
    arr_full, total_full = model_fn(" ".join(words), target_text)
    T = len(arr_full)
    accumulator = np.zeros((n, T))

    def v_of_subset(sorted_tuple):
        if sorted_tuple in cache:
            return cache[sorted_tuple]
        prompt_text = prompt_from_indices(words, sorted_tuple)
        arr, _ = model_fn(prompt_text, target_text)
        cache[sorted_tuple] = arr
        return arr

    for _ in tqdm(range(num_permutations), desc="Shapley perms"):
        perm = list(range(n))
        random.shuffle(perm)
        S = []
        for idx in perm:
            S_sorted = tuple(sorted(S))
            S_with = tuple(sorted(S + [idx]))
            vS = v_of_subset(S_sorted)
            vSi = v_of_subset(S_with)
            marginal = vSi - vS
            accumulator[idx] += marginal
            S.append(idx)

    return accumulator / float(num_permutations)

print("\nComputing Shapley contributions...")
shap_matrix = approximate_shapley_sentence(prompt_words, output_text, co
mpute_logprobs_for_prompt)

# ---------------------------
# STEP 5: Aggregate by output sentences
# ---------------------------
def aggregate_by_sentences(shap_matrix, output_text, tokenizer, prompt_labe
ls, agg="sum_abs"):
    sentences = [s.strip() for s in output_text.split(".") if s.strip()]
    token_counts = [len(tokenizer(s).input_ids) for s in sentences]
    idx = 0
    ranges = []
    for c in token_counts:
        ranges.append((idx, idx + c))
        idx += c

    n_words, _ = shap_matrix.shape
    n_sents = len(ranges)
    agg_mat = np.zeros((n_words, n_sents))
    for si, (a, b) in enumerate(ranges):
        if agg == "sum_abs":
            agg_mat[:, si] = np.abs(shap_matrix[:, a:b]).sum(axis=1)
        else:
            agg_mat[:, si] = shap_matrix[:, a:b].sum(axis=1)
    return agg_mat, sentences

agg_mat, sents = aggregate_by_sentences(shap_matrix, output_text, tokenize
r, prompt_words)

# ---------------------------
# STEP 6: Visualize as heatmap
# ---------------------------
plt.figure(figsize=(12, max(3, len(prompt_words) * 0.25)))
sns.heatmap(
    agg_mat,
    xticklabels=[(s[:60] + "...") if len(s) > 60 else s for s in sents],
    yticklabels=prompt_words,
    cmap="RdYlGn",
    center=0.0
)
plt.title("Prompt → Output Sentence Influence (Shapley values)")
plt.xlabel("Output Sentences (truncated)")
plt.ylabel("Prompt Words")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()
```



🧠 Recap of What This Code Does (High-Level)

Here's what happens step by step — explained like a human, not a manual:

1. **Prompt setup** — you define the question you want to test.
2. **Hardcoded output** — you use a fixed response (so we don't waste GPU cycles generating).
3. **Log-probability computation** — you run the model on (prompt + output) and get the log probability for each token of the output.
4. **Shapley estimation** — you test what happens when you remove or keep subsets of prompt words, and see how much each word's absence changes output token probabilities.
5. **Aggregate by sentence** — you group tokens into sentences and compute the *total effect* of each prompt word on each output sentence.
6. **Heatmap visualization** — this gives a "Prompt word → Output sentence" influence matrix.

So it answers:

> 🌟**"If I remove or change this word in the prompt, which sentences in the output are most affected?"**