Fine-Tuning Large Language Models for Enterprise Applications

Use Case #2: Medical Misinformation Detection in LLM Responses

Project Report

---

1. Student Information

- **Student Name:** Adeel Ahmed
- **Student ID:** 224404186
- **Date Submitted:** 16-May-2025

---

2. Project Introduction

- **Title of the Project:**

  Medical Misinformation Detection in LLM Responses

- **What is the project about?**

  This project fine-tunes an open-source LLM (Mistral 7B) using Low-Rank Adaptation (LoRA) to classify medical claims as **true**, **false**, or **misleading**.

- **Why is this project important or useful?**

  AI-generated responses can confidently propagate incorrect or misleading health advice and pose real risks. By equipping an LLM to detect such misinformation, we can flag or filter harmful content in medical chatbots, summaries, or social-media monitoring tools, thereby improving safety and trust in AI-powered healthcare applications.

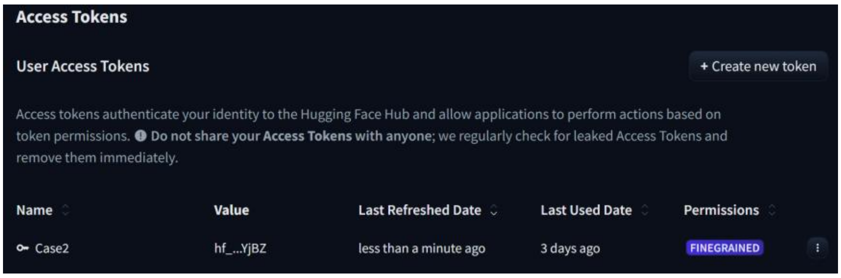
---

3. API/Token Setup — Step-by-Step

Provider: Hugging Face (for model and dataset access)

Steps to generate the token

- **Step 1:** Create a Hugging Face account at https://huggingface.co
- **Step 2:** Click your avatar → Settings → Access Tokens
- **Step 3:** Click New token, give it a name (“Case2”), and grant read scope
- **Step 4:** Copy the token (hf_...) and save it securely



Secure Loading of Token in Code:
Avoid hardcoding your token in the notebook. Mount your Drive, read the token from a private file, and inject it into os.environ before calling login():

Environment Setup

```
[ ] from google.colab import drive
    drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Imports & GPU Check

```
import torch
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling
)
import datasets
import peft
import numpy as np
import os

from huggingface_hub import login

# Load hf_token from Google Drive
token_path = "/content/drive/MyDrive/SIT764 - Case 2/hf_token.txt"
with open(token_path, "r") as f:
    os.environ["HF_TOKEN"] = f.read().strip()

login(token=os.environ["HF_TOKEN"], add_to_git_credential=True)

print("Hugging Face login successful!")

# Check GPU
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using device:", device)
```

```
Hugging Face login successful!
Using device: cuda
```

---

4. Environment Setup

Development Platform: Google Colab

GPU Available? Yes

GPU Type: NVIDIA A100-SXM4-40GB

Python Version: 3 (Colab default)

Other Tools: Jupyter notebook interface, Google Drive for persistent storage

Code: Environment & GPU Check

Environment Setup

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
[ ] !nvidia-smi
```

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA A100-SXM4-40GB          Off |   00000000:00:04.0 Off |                    0 |
| N/A   34C    P0             43W /  400W |       0MiB /  40960MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
```

---

5. LLM Setup

Model: mistralai/Mistral-7B-v0.1 (7-billion parameter open-source model on Hugging Face).

Provider: Hugging Face (model and tokenizer).

LoRA: Applied Low-Rank Adaptation (LoRA) to inject trainable adapters while keeping the base model weights frozen. LoRA greatly reduces tuning cost for large models. (See my **WHY LoRA?** Documentation in company team channel)

Libraries & Dependencies:

- **transformers (v4.x)** for model/tokenizer classes and training infrastructure.
- **peft (v0.x)** for LoRA integration.
- **datasets** for loading/preprocessing.
- **torch (PyTorch)** for tensors and GPU training.
- **bitsandbytes** (used for quantization in SciFact fine-tuning stage)
- **numpy, scikit-learn** (for metrics), **wandb** (for logging).

Quantization Details (bitsandbytes):
To efficiently fine-tune the Mistral-7B model on SciFact within GPU memory constraints, **4-bit quantization (NF4 + double quantization)** was applied using bitsandbytes. The specific configuration used is:

Load the model

```
[ ] from transformers import AutoModelForSequenceClassification, BitsAndBytesConfig

    base_model_name = "mistralai/Mistral-7B-v0.1"
    # 4-bit Quantization config (NF4 + double quant)
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.float16,
    )

    # 4. Load & quantize base model
    model = AutoModelForSequenceClassification.from_pretrained(
        base_model_name,
        quantization_config=bnb_config,
        device_map="auto",
        torch_dtype=torch.float16,
        num_labels=3,
        problem_type="single_label_classification",
    )
    model.config.pad_token_id = tokenizer.eos_token_id
```

---

6. Dataset Description

I used three medical fact-checking datasets, each reformatted for a 3-class classification problem (labels: "true," "false," "misleading"):

- **HealthFact:** Medical claims with associated evidence, explicitly labeled true, false, or misleading.
    - **Files:** healthfact_train.json, cleaned_healthfact_dev.json, cleaned_healthfact_test.json
- **SciFact:** Scientific claims and evidence pairs from a scientific fact-checking benchmark dataset.
    - **Files:** train_3class.jsonl, dev_3class.jsonl
- **COVID-19:** COVID-related claims from social media and news sources, labeled true or false.
    - Files available but not used in training or evaluation for this implementation.

Each dataset record has the following fields:

- **text:** Claim being evaluated
- **evidence text or explanation:** Supporting evidence or explanation (optional)
- **label:** Ground-truth classification label (true, false, misleading)

Preprocessing:

I applied consistent preprocessing across all datasets, including COVID-19 (for completeness), but did not include the COVID-19 dataset in the actual training or evaluation process.

- **Data loading code (example for HealthFact):**

```
health_dataset = datasets.load_dataset("json", data_files={
    "train": "/content/drive/MyDrive/SIT764 - Case 2/HealthFact/healthfact_traindata.json",
    "validation": "/content/drive/MyDrive/SIT764 - Case 2/HealthFact/cleaned_healthfact_dev.json",
    "test": "/content/drive/MyDrive/SIT764 - Case 2/HealthFact/cleaned_healthfact_test.json"
})
```

- **Rename fields:** Ensure each dataset has a column text (for the claim) and labels. For instance, original columns like tweet or claim were standardized to text.

```
# Rename 'claim' -> 'text'
# Keep 'explanation' for later
health_dataset = health_dataset.rename_column("claim", "text")
health_dataset = health_dataset.rename_column("label", "labels")
```

- **Label encoding:** Map label strings to integers:

```
# Convert Labels to Integers
label2id = {
    "false": 0,
    "true": 1,
    "misleading": 2
}
```

- **Prompt formatting:** Merge claim and evidence into a single input. For example:

```
# Function to merge claim and evidence into a single text field
def format_input(example):
    evidence = example.get("evidence_text") or example.get("explanation") or "(none)"
    return {
        "text": f"Claim: {example['text']}\\nEvidence: {evidence}",
        "labels": label2id[example["labels"].lower()]
    }
```

I applied this mapping to each dataset split. The final model input is thus a string like "Claim: ... Evidence: ...", and the corresponding labels tensor.

- **Tokenization:** I used the Hugging Face tokenizer associated with the 'mistralai/Mistral-7B-v0.1' model. Tokenization transforms text inputs into numeric representations (input_ids) that the model can process. Since the Mistral tokenizer does not define a padding token (pad_token) by default, I explicitly set it to the tokenizer's eos_token:

Tokenize the HealthFact Dataset

```
[ ] model_name = "mistralai/Mistral-7B-v0.1"
    tokenizer = AutoTokenizer.from_pretrained(model_name) # Creating Tokenizer
    # Mistral doesn't define a pad_token by default, so we reuse the eos_token as pad_token
    if tokenizer.pad_token is None:
        tokenizer.pad_token = tokenizer.eos_token
        tokenizer.pad_token_id = tokenizer.eos_token_id

    def tokenize_function(examples):
        return tokenizer(examples["text"], padding="max_length", truncation=True, max_length=128)

    health_tokenized = health_dataset.map(tokenize_function, batched=True)
```

```
# Convert Labels to Integers
label2id = {
    "false": 0,
    "true": 1,
    "misleading": 2
}
def encode_labels(example):
    example["labels"] = label2id[example["labels"].lower()]
    return example

health_tokenized = health_tokenized.map(encode_labels)

sample = health_tokenized["train"][0]
print("Labels:", sample["labels"], type(sample["labels"]))
```

This produced consistent input tensors (input_ids, attention_mask, labels) ready for training.

Class Distribution & Imbalance:

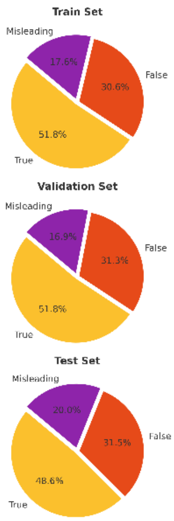
I analysed the label distribution across both **HealthFact** and **SciFact** datasets. All datasets exhibit moderate imbalance, with the "true" class making up over 50% and "misleading" the minority class (17%)

HealthFact example:

```
HealthFact Train label distribution:
label
true          0.517952
false         0.306100
MISLEADING    0.175949
Name: proportion, dtype: float64

HealthFact Validation label distribution:
label
true          0.518122
false         0.313015
MISLEADING    0.168863
Name: proportion, dtype: float64

HealthFact Test label distribution:
label
true          0.485807
false         0.314680
MISLEADING    0.199513
Name: proportion, dtype: float64
```



Observation:

- The "misleading" class is underrepresented in both datasets.
- This can result in poor recall for that class unless handled properly.

Handling Class Imbalance

The following strategies were applied:

Class Weighting:
I used compute_class_weight from scikit-learn to dynamically adjust loss weights for each class based on their frequency in the training data. This ensured the model paid more attention to minority classes, especially "misleading".

- **Custom Trainer:**
  A modified Hugging Face Trainer subclass was used to apply a weighted CrossEntropyLoss using the computed class weights.
- **Macro & Weighted F1 Evaluation:**
  Both metrics were used:
  - Macro F1 to measure how well the model performs on each class equally.
  - Weighted F1 to reflect overall performance considering class proportions.

Note: See my "**Handling Class Imbalance – Quick Guide**" document in company team channel for more information.

---

7. Improving LLM Performance

I gradually improved the model through several stages. The table below summarizes each step, its method, and the key benchmark result (accuracy and weighted F1 score on the HealthFact test set):

Step	Method	Description	Test Accuracy	Weighted F1
1	Baseline	Evaluate base Mistral-7B on HealthFact test set without any fine-tuning.	41.2%	37.3%
2	Fine-tune (HealthFact)	Apply LoRA fine-tuning on HealthFact training set (3 epochs).	57.2%	56.8%
3	Fine-tune (SciFact)	Continue training the same LoRA adapter on SciFact dataset after HealthFact. (3 epochs)	55.7%	55.5%
4	Combined Fine-tune	Train LoRA adapters from scratch on combined HealthFact+SciFact datasets. (3 epochs)	65.4%	61.7%
5	Prompt-Engineered Fine-Tuning	Same combined dataset but using prompt-formatted inputs (Claim: ...\nEvidence: ...).	78.1%	77.8%

Each step used the Hugging Face **Trainer** with appropriate **TrainingArguments**. For example, after loading the base model and tokenizer, I wrapped it with LoRA adapters:

Set up LoRA configuration

```python
from peft import LoraConfig, get_peft_model, TaskType

lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.0,
    bias="none",
    task_type=TaskType.SEQ_CLS
)

model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
```

```
trainable params: 3,420,160 || all params: 7,114,092,544 || trainable%: 0.0481
```

Code Snippets for Each Step:

Step 1 (Baseline): Load model/tokenizer and evaluate on HealthFact test set (no training). Example code:

```python
# 1. Load the base model (pre-finetuning)
base_model_name = "mistralai/Mistral-7B-v0.1"
tokenizer = AutoTokenizer.from_pretrained(base_model_name)
base_model = AutoModelForSequenceClassification.from_pretrained(
    base_model_name,
    device_map="auto",
    torch_dtype=torch.float32,
    num_labels=3,
    problem_type="single_label_classification"
)
# Reuse the eos_token as pad_token
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
    tokenizer.pad_token_id = tokenizer.eos_token_id
base_model.config.pad_token_id = tokenizer.pad_token_id
base_model.eval()

# 2. Use the already tokenized test dataset (health_tokenized["test"])
test_dataset = health_tokenized["test"]

# Filter Out Unused Columns
cols_to_keep = {"input_ids", "attention_mask", "labels"}
cols_to_remove = list(set(test_dataset.column_names) - cols_to_keep)
test_dataset = test_dataset.remove_columns(cols_to_remove)

# Convert dataset fields to PyTorch tensors
test_dataset.set_format(
    type="torch",
    columns=["input_ids", "attention_mask", "labels"]
)

test_loader = DataLoader(test_dataset, batch_size=8)
```

```python
# 3. Inference Loop
all_preds, all_labels = [], []
for batch in test_loader:
    # Assuming batch fields are already tensors
    batch = {k: v.to("cuda") for k, v in batch.items()}
    with torch.no_grad():
        outputs = base_model(**batch)
    preds = torch.argmax(outputs.logits, dim=-1)
    all_preds.append(preds.cpu())
    all_labels.append(batch["labels"].cpu())

all_preds = torch.cat(all_preds).numpy()
all_labels = torch.cat(all_labels).numpy()

# 4. Calculate metrics
cm = confusion_matrix(all_labels, all_preds)
acc = accuracy_score(all_labels, all_preds)
f1 = f1_score(all_labels, all_preds, average="weighted")

print("Test Accuracy (Pre-finetuning):", acc)
print("Weighted F1 Score (Pre-finetuning):", f1)
print("Confusion Matrix (Pre-finetuning):\n", cm)
```

Step 2 and 3 (HealthFact Fine-tune and then SciFact continuation):

As shown above, I applied LoRA (LoraConfig) and trained with Trainer.train().

Define Training Arguments & Metrics

```
[ ] import numpy as np
    from sklearn.metrics import accuracy_score, f1_score
    from transformers import TrainingArguments, Trainer

    checkpoint_dir = "/content/drive/MyDrive/SIT764 - Case 2/HealthFact/content/health_finetuned_lora"

    def compute_metrics(eval_pred):
        logits, labels = eval_pred #logits are raw model outputs
        preds = np.argmax(logits, axis=-1) # Converts logits into predicted class indices (predicted labels)
        acc = accuracy_score(labels, preds)
        f1 = f1_score(labels, preds, average="weighted")
        return {"eval_accuracy": acc, "eval_f1": f1}

    training_args = TrainingArguments(
        output_dir="/content/drive/MyDrive/SIT764 - Case 2/SciFact/SciFactResults/scifact_results",
        overwrite_output_dir=True,
        eval_strategy="steps",
        save_strategy="steps",
        eval_steps=100,
        save_steps=200,
        num_train_epochs=3,
        per_device_train_batch_size=4,
        per_device_eval_batch_size=4,
        logging_steps=10,
        bf16=False,             # disable half precision, use fp32 (full precision)
        learning_rate=1e-6,     # lower LR
        lr_scheduler_type="cosine",
        max_grad_norm=1.0,
        label_names=["labels"],
        load_best_model_at_end=True,
        metric_for_best_model="eval_f1",
        resume_from_checkpoint = checkpoint_dir
    )
```

```
trainer_health = Trainer(
    model=health_model,
    args=training_args_health,
    train_dataset=health_tokenized["train"],
    eval_dataset=health_tokenized["validation"],
    compute_metrics=compute_metrics
)
```

```
# Initialize Trainer and fine-tune on SciFact
trainer = Trainer(
    model=lora_model,
    args=training_args,
    train_dataset=sci_data["train"],
    eval_dataset=sci_data["validation"],
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics
)

trainer.train()
```

Step 4 (Combined Fine-tune): To mitigate catastrophic forgetting observed in staged fine-tuning (step 2 and 3), I started training LoRA adapters on both datasets simultaneously from the start.

What was done:

- Merged HealthFact and SciFact datasets.
- Tokenized and labeled using a shared format.
- Computed **class weights** using sklearn.utils.class_weight.compute_class_weight to address label imbalance (e.g. “true” class dominant).
- Used a **custom Trainer** that applies these class weights during loss computation.

Compute Class Weights and Define Custom Loss

```
from sklearn.utils.class_weight import compute_class_weight
import numpy as np
import torch
from torch.nn import CrossEntropyLoss


all_labels = np.array(combined_train_tokenized["labels"]).astype(int)
class_weights = compute_class_weight(class_weight="balanced", classes=np.unique(all_labels), y=all_labels)
class_weights_tensor = torch.tensor(class_weights, dtype=torch.float).to(model.device)

loss_fn = CrossEntropyLoss(weight=class_weights_tensor)
```

Define a Custom Trainer with Weighted Loss

```
class CustomTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False, **kwargs):
        labels = inputs.pop("labels")
        outputs = model(**inputs)
        logits = outputs.logits
        loss = loss_fn(logits, labels)
        return (loss, outputs) if return_outputs else loss
```

Why it matters:
Combining datasets preserved knowledge from both domains. Applying class weights ensured that underrepresented classes (like "misleading") were not ignored during training.

Step 5 (Prompt-Engineered Fine-Tuning): Leverage improved prompt design (explicit Claim: ... and Evidence: ... structure) to guide model understanding.

What was changed:

- No change in datasets or model architecture.
- Only the input format was restructured to clearly separate and label the claim and evidence fields.

Why it worked:
The clearer structure helped the model disambiguate context, especially for harder cases like “misleading” vs “false.”

---

8. Benchmarking & Evaluation

Metrics Used

- **Accuracy:** Measures overall prediction correctness across all classes.
- **Macro F1:** Evaluates each class independently and equally, ideal for imbalanced multi-class classification.
- **Weighted F1:** Averages F1 scores by class frequency, giving a realistic view of overall performance under imbalance.

Why These Metrics?

These metrics are well-suited for **multi-class classification** tasks, especially when class distribution is imbalanced:

- **Macro F1** treats all classes equally, helping assess fairness across "true", "false", and "misleading" labels.

- **Weighted F1** reflects overall model performance while accounting for the actual class distribution.

Together, they provide a balanced view of performance across both majority and minority classes.

Benchmark Dataset & Sample Size

- **Benchmark Dataset**: cleaned_healthfact_test.json
- **Sample Size**: 1,233 labeled examples
- Used consistently to evaluate all model stages.

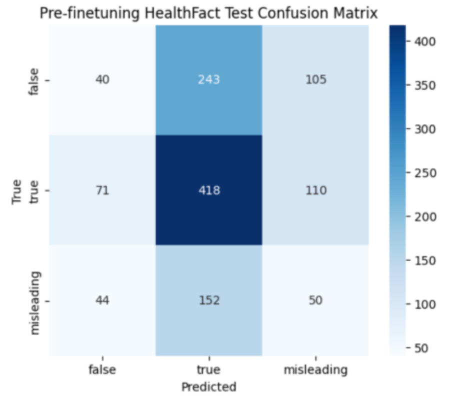
Code: Metric Calculation

```python
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=-1)
    return {
        "eval_accuracy": accuracy_score(labels, preds),
        "eval_f1": f1_score(labels, preds, average="macro"),
        "macro_f1": f1_score(labels, preds, average="macro"),
        "weighted_f1": f1_score(labels, preds, average="weighted"),
    }
```

Plot Results

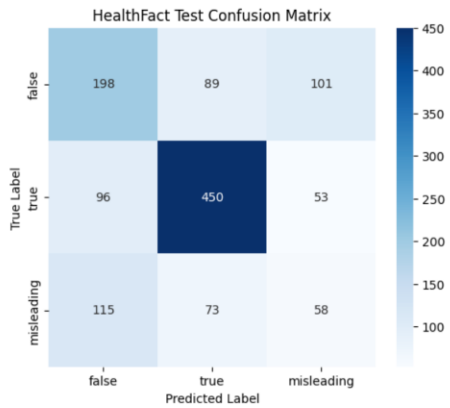
Step 1 (Baseline):

```
Test Accuracy (Pre-finetuning): 0.4120032441200324
Weighted F1 Score (Pre-finetuning): 0.37303611487744265
Confusion Matrix (Pre-finetuning):
 [[ 40 243 105]
 [ 71 418 110]
 [ 44 152  50]]
```



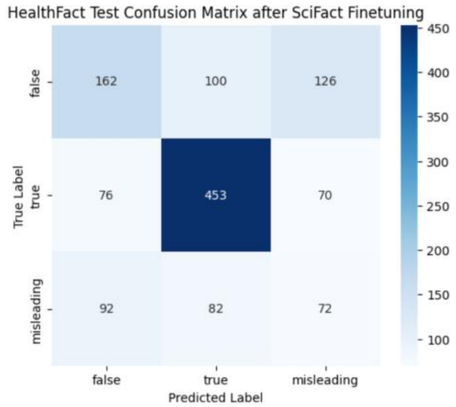
Step 2 (HealthFact Fine-tune):

```
Test Accuracy: 0.5725871857258719
Weighted F1 Score: 0.56793016581622
Confusion Matrix:
 [[198  89 101]
 [ 96 450  53]
 [115  73  58]]
```



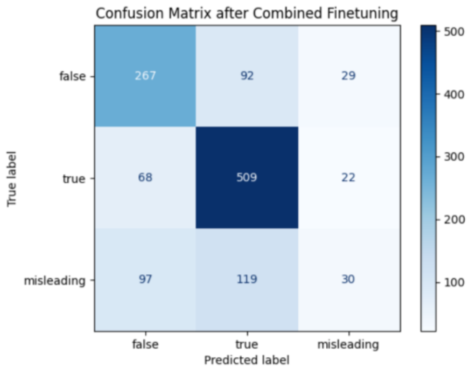
Step 3 (SciFact Fine-tune):

```
Test Accuracy: 0.5571776155717761
Weighted F1 Score: 0.5545734628998064
Confusion Matrix:
 [[162 100 126]
 [ 76 453  70]
 [ 92  82  72]]
```



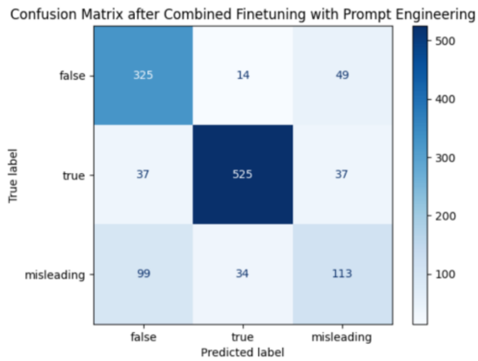
Step 4 (Combined Fine-tune):

```
Accuracy      : 0.6537
Macro F1      : 0.5355
Weighted F1   : 0.6165
```



Step 5 (Prompt-Engineered Fine-Tuning):

```
Accuracy      : 0.7810
Macro F1      : 0.7231
Weighted F1   : 0.7775
```



Interpretation

- Biggest gain came from prompt refinement (+13% from combined stage).
- Fine-tuning on HealthFact alone brought a strong initial lift (+16% from zero-shot).
- SciFact continuation slightly degraded performance (catastrophic forgetting).
- Combined fine-tuning improved balance, but structure refinement made the biggest difference.

---

9. UI Integration

Tool Used

- **Frontend:** React
- **Backend:** FastAPI
- **Deployment:** Docker, Kubernetes (with separate services for frontend and backend)

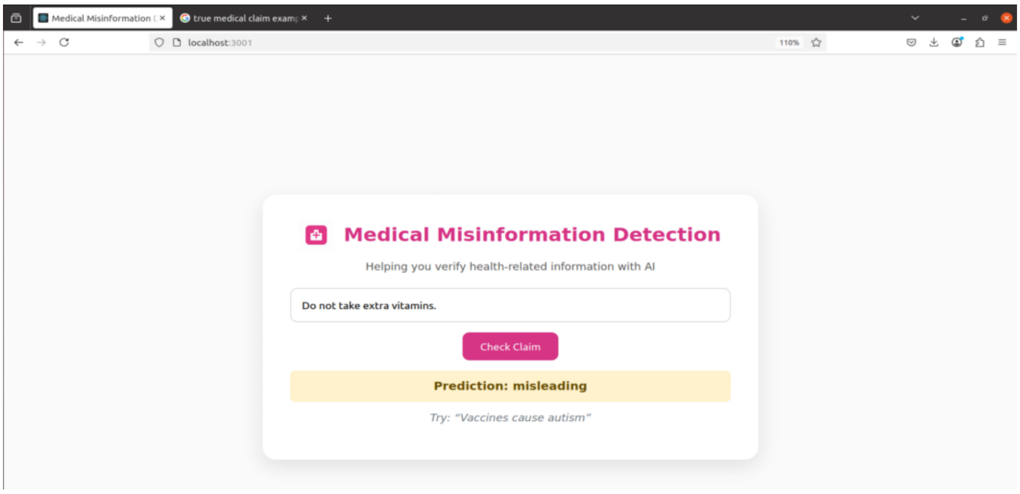
Key Features of the Interface

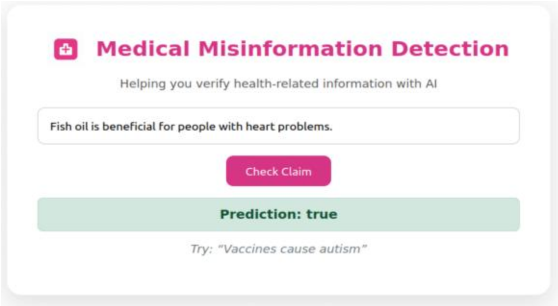
- Simple web UI where users can input medical claims and receive real-time classification as true, false, or misleading.
- Frontend is responsive and cleanly structured for clarity and ease of use.
- Backend handles requests via FastAPI and routes them to a fine-tuned Mistral-7B model running on Colab.
- Uses REST API endpoints for interaction between frontend and backend.
- Fully containerized and orchestrated in Kubernetes for scalability and modularity.

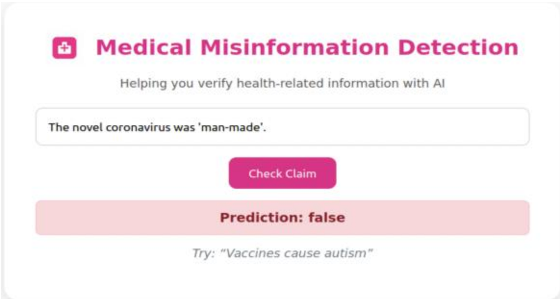
How It Works

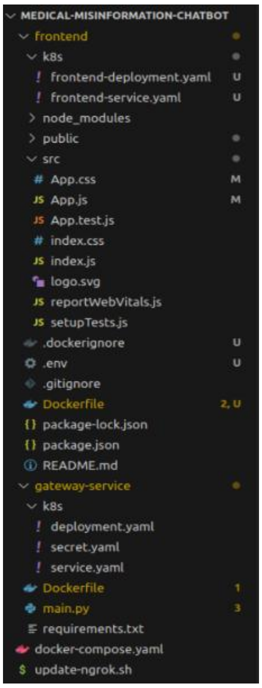
- User enters a claim on the frontend.
- React sends a POST request to the FastAPI backend (/predict endpoint).
- Backend forwards the request to the model and returns the predicted label.
- Result is displayed on the UI.

Screenshots of Working UI









Example Snippet: FastAPI Inference Endpoint

FastAPI App for Prediction

```python
app = FastAPI()

class Query(BaseModel):
    question: str

label_map = {0: "false", 1: "true", 2: "misleading"}

@app.post("/predict")
def predict(query: Query):
    inputs = tokenizer(
        query.question,
        return_tensors="pt",
        truncation=True,
        padding=True,
        max_length=128
    ).to("cuda")

    with torch.no_grad():
        outputs = model(**inputs)
    pred_class = torch.argmax(outputs.logits, dim=-1).item()
    return {
        "label": label_map[pred_class],
        "class_id": pred_class
    }
```