Decoding Diner Safety: Summarizing & Extracting Keywords from Chicago's Restaurant Health Violations to Improve Customer Health Awareness

Group Members: Carolina Aldana Yabur, Shlok Nandkishor Goud, Zahid Rahman

ISM 6564 Text Analytics

Professor Anol Bhattacherjee

7 May 2025

School of Information Systems and Management - Muma College of Business -

University of South Florida

(1) Executive Summary

Dining out in Chicago can be risky: while the city inspects over 200,000 food establishments each year, inspection results are only available as lengthy tables and detailed narratives that overwhelm most customers. To bridge this gap, our team built a web-based tool that translates raw inspection records into brief, plain-language summaries, highlights key safety concerns, and offers a simple restaurant safety rating.

We sourced authentic inspection data directly from the City of Chicago's public health API and cleaned it to remove duplicates and irrelevant entries. On the back end, we fine-tuned a lightweight transformer model (Flan-T5 Small with Low-Rank Adaptation) to generate concise violation summaries and extract issue tags such as "rodents," "expired food," or "staff hygiene." A Flask API serves these results, and a React front end delivers them to users with a one-click search interface.

While our prototype successfully integrates data ingestion, model inference, and user interaction, evaluation revealed that the summaries often suffered from inconsistency—blank outputs, repetitive phrasing, or missing context—likely due to the model's limited capacity and training data volume. This suggests that, in its current form, the tool is not yet reliable for critical consumer decisions.

Looking ahead, upgrading to a more capable language model (e.g., GPT-3.5 or Mistral-7B), incorporating retrieval-augmented generation to ground summaries in actual inspection text, and precomputing results for faster, uniform responses will be key. With these enhancements, this system can evolve into a robust, user-friendly platform that empowers Chicago diners to make safer choices and encourages restaurants to maintain higher food-safety standards.

(2) Problem Definition & Significance

Every year, more than 60% of foodborne illness outbreaks stem from unsafe practices in restaurants, putting millions of diners at risk of serious health problems (Shafieizadeh et al., 2023). In Chicago alone, the Department of Public Health inspects over 200,000 food establishments annually, but its detailed results—stored as long tables and narrative reports—are hard for the average customer to digest quickly. As a result, diners rely on word-of-mouth reviews rather than official data, leaving them blind to patterns of critical violations such as rodent sightings or improper food storage.

Our target users are Chicago restaurant patrons who want clear, reliable safety information before choosing where to eat. By turning raw inspection records into a concise safety summary, issue tags, and an overall "safety rating," we empower decision-makers—families, tourists, and busy professionals—to evaluate risk at a glance. For restaurant owners and public-health officials, the tool offers a transparent feedback loop, highlighting recurring problems and incentivizing improvements. Enabling even a fraction of Chicago's estimated 1 million annual diners to spot high-risk venues could meaningfully reduce foodborne illnesses and drive higher compliance across the industry

(3) Prior Literature

Researchers have applied machine learning to flag high-risk restaurants before inspections. Kang et al. (2013) matched 150,000 Yelp reviews to Seattle inspection records and trained support-vector classifiers that identified severe hygiene violations with over 82% accuracy. Mejia et al. (2019) combined 1.3 million Yelp reviews with official New York City inspection data to detect post-inspection declines in cleanliness, showing that 30% of restaurants backslide within 90 days by monitoring crowd-sourced keywords. Kannan et al. (2019) audited the Chicago Department of Public Health's own forecasting model and found it cut identification time of critical violations by 7.4 days, though it introduced inspector-bias and relied on limited features. While these studies demonstrate the power of predictive analytics and social-media signals, they focus on internal agency tools rather than consumer-facing insights.

In parallel, text-summarization techniques have proven effective at distilling long documents. Dumne et al. (2024) used TextRank—an extractive graph-based algorithm—to condense complex reports with about 90% accuracy in key-sentence selection. Mahmood and Khan (2019) applied classification methods (SVM, Naïve Bayes, Random Forest) to New York City inspection narratives, revealing that simple feature selection techniques can reliably surface the most common violation themes. Schomberg et al. (2016) extended this idea by leveraging Twitter and Yelp keywords to act as early warnings, achieving 91% sensitivity in San Francisco pilot studies.

Despite these advances, no existing work offers real-time, plain-language summaries of official inspection reports directly to consumers. Our project goes beyond prior approaches by fine-tuning a transformer model to generate concise violation summaries and extract standardized issue tags, all served through a one-click web interface. This combination of summarization and keyword extraction fills a critical gap: giving everyday diners an immediate, interpretable view of restaurant safety records.

(4) Data Source & Preparation

In mid-April 2025, we obtained the City of Chicago's Food Inspections export (Food_Inspections_20250419.csv) from the official Data Portal, which catalogs over 200,000 inspection records—including violation narratives, dates, facility IDs, and severity codes. We filtered out entries missing narratives, normalized date fields, and drew a stratified random sample of 3,000 records to ensure coverage across all severity levels. These were partitioned into training (80 %), validation (10 %), and test (10 %) splits.

Next, we enriched each split via a scalable AI-powered workflow. Records were sent in parallel to the OpenAI Chat API (gpt-3.5-turbo) with a prompt that returned precisely three elements: a consumer-friendly summary, 3–7 keywords, and one of four safety verdicts. Robust retry/backoff logic handled rate limits, and any failed or malformed responses were dropped. To bolster examples of serious violations and address class imbalance, we generated 100 synthetic "unsafe to eat here" narratives and processed them identically. Finally, real and synthetic records were merged, shuffled, and saved as JSONL files (`violations_train_final.jsonl`, `violations_val.jsonl`, `violations_test.jsonl`). Automated checks ensured every summary ended with a valid verdict, and manual spot-checks confirmed output coherence. (Full enrichment prompts and code are in the Appendix.)

(5) Methodology

Building on our enriched data, we implemented a three-tier NLP solution:

1. **Adapter-Based Model Fine-Tuning**
   Zahid loaded Google's Flan-T5-Small in 4-bit NF4 format and applied lightweight adapter modules (rank 16, $\alpha = 32$, dropout 0.05), updating only these adapters during training. Using Hugging Face's Seq2SeqTrainer, he ran three epochs with batch size 8, a peak learning rate of 2×10^{-5}, and BF16 precision. The model was optimized against the JSONL targets, and the best checkpoint was chosen based on validation loss.
2. **Backend API Deployment**
   Shlok wrapped the fine-tuned model in a Flask service exposing a /search_and_summarize endpoint. For each restaurant query, the API fetches live inspection data from Chicago's Socrata API, cleans the narratives, and invokes the model to return structured {summary, keywords, verdict} JSON. Caching and greedy decoding (max 128 tokens) keep end-to-end latency under one second in a Dockerized environment.
3. **Frontend Application**
   Carolina developed a React single-page interface with Material UI. Users enter a restaurant name or address and see a loading spinner during processing. Results display a concise safety summary and keyword tags; any lookup errors trigger clear, dismissible banners. The UI was refined through both simulation mode (with mock data responses) and live-mode testing against real queries.

This modular architecture ensured a smooth flow from raw inspection data to user-focused insights. Detailed model configurations, prompt templates, and deployment scripts appear in the Appendix.

(6) Experiments & Results

Our evaluation revealed that, while individual components functioned as intended, a critical misalignment between our enrichment targets and model inference undermined end-to-end reliability.

Training Convergence
Over three epochs, training and validation loss steadily declined, demonstrating effective adaptation of the LoRA adapters (Table 1).

Epoch	Training Loss	Validation Loss
1	3.3185	2.8308
2	2.8187	2.2999
3	2.6515	2.1645

Automatic Metrics
On 300 held-out test records, fine-tuning improved summary quality over the base model, particularly in ROUGE scores, though BERTScore saw a slight decline (Table 2). Keyword extraction—evaluated only for the fine-tuned model—achieved precision ≈ 0.65, recall ≈ 0.60, and F1 ≈ 0.62.

Metric	Base Model	Fine-Tuned Model
ROUGE-1	0.1288	**0.1554**
ROUGE-2	0.0317	**0.0372**
ROUGE-L	0.1005	**0.1216**
BERTScore F1	0.4653	0.4346

Note: Keyword extraction results (Fine-Tuned)—Precision: 0.65; Recall: 0.60; F1: 0.62.

Critical Robustness Issue

Despite successful convergence, the model only produces valid JSON when prompted with the *exact* enrichment instruction used during training. Any deviation—such as the shorter "Summarize this…" prompt—caused 15 % of outputs to fail parsing (empty or malformed). This brittleness represents a fundamental misstep: our enrichment pipeline and model inference were not aligned, so downstream components (API, UI) could not reliably consume model outputs.

User Experience Breakdown

- **Simulation mode**: The UI gracefully handled mock responses, returning a generic "sample data" summary with `["mock data", "safe", "restaurant"]`.
- **Live mode**: Queries against real Chicago data exposed jarring failures:
  - Summaries were frequently vague or nonsensical.
  - Keyword arrays missing entirely.
  - Some valid addresses yielded **404 – NOT FOUND** errors.

These issues trace directly to the parser failures above—if the model didn't emit valid JSON, the UI could neither display keywords nor summaries, breaking the user flow.

Latency & Throughput

End-to-end timing—fetching Socrata records, model inference, and JSON parsing—averaged **820–950 ms** per query under typical loads.
Instrumented the pipeline with **lightweight logging and response timers**, enabling empirical tracking of each stage in the request lifecycle. These measurements confirmed that most of the latency stemmed from model inference, while Socrata API fetch times varied slightly based on query specificity. The Flask app's stateless design and token-limited greedy decoding ensured consistent performance across repeated queries, making the system responsive enough for real-time use during testing.

(7) Insights & Future Work

Code Recommendations

Zahid (Data Enrichment & Model Fine-Tuning)

- **Prompt-Robust Training**: Include multiple prompt formulations during fine-tuning so the model generalizes beyond a single, rigid instruction.
- **Adapter-Loading Fix**: Resolve the bug that causes LoRA adapters to fall back to the base model at inference.
- **Model Upgrade**: Swap Flan-T5-Small for a larger instruction-tuned LLM (e.g. GPT-3.5 or Mistral-7B) and retrain on the full corpus plus synthetic examples.
- **Synthetic Data Balance**: Expand synthetic "unsafe" examples or diversify their scenarios to improve recall on severe violations.

Shlok (Flask API Integration)

- **Prompt Prefixing**: Always prepend the exact JSON-generation prompt to each inference call to guarantee valid `{"summary", "keywords", "verdict"}` output.
- **Offline Caching**: Precompute and store summaries/keywords in a database to eliminate real-time generation failures and reduce latency.
- **Enhanced Error Handling**:
  - Catch and surface Socrata API errors (e.g. 404) with retry/backoff or graceful fallbacks.
  - Validate parsed JSON before returning to the UI and log parse failures for monitoring.
- **Instrumentation**: Add request logging and metrics (latency, error rates, JSON-success percentage) to a dashboard for ongoing health checks.

Carolina (React Frontend UI)

- **Mode Awareness**: Clearly indicate "simulation" vs. "live" mode so users understand mock data behavior.
- **Resilient Rendering**: Provide fallback UI (placeholder text or default tags) when keywords or summary fields are missing.
- **Error Messaging**:
  - Make "API Error: 404 – NOT FOUND" banners more descriptive (e.g. "No inspections found for that address").
  - Offer retry or "search again" actions directly in the banner.
- **Loading & Empty States**: Enhance spinners and empty-state screens to guide users on next steps if no data is returned.

Non-Code (Overall Project) Recommendations

1. **Clarify Product Vision**
    a. Decide if the app's core is risk alerts, raw-data access, or concise summaries—then align UX, metrics, and stakeholder goals accordingly.
2. **Shift to Offline Precomputation**
    a. Batch-generate and cache all summaries/keywords nightly to guarantee consistent, sub-second responses.
3. **Strengthen Model Robustness**
    a. Retrain on a larger LLM and incorporate prompt-variation strategies to cut JSON-failure rates below 5 %.
4. **Enhance Evaluation & Monitoring**
    a. Build a dashboard tracking JSON-parse success, latency, ROUGE/BERTScore trends, and UI error frequencies.
    b. Conduct regular manual spot-checks of random summaries for quality assurance.
5. **Localize & Personalize**
    a. Add multi-language support (e.g., Spanish, Polish) and filtering by neighborhood or time frame to broaden user appeal.
6. **Integrate with Public Health Ecosystem**
    a. Publish an open API of structured violation summaries and keywords to partner with delivery apps, tourism sites, and city portals for wider impact.

(8) References

Dumne, R., Gavankar, N. L., Bokare, M. M., & Waghmare, V. N. (2024). Automatic Text Summarization using Text Rank Algorithm. 2024 3rd International Conference for Advancement in Technology (ICONAT), 1–6. https://doi.org/10.1109/ICONAT61936.2024.10775241

Kang, J. S., Kuznetsova, P., Luca, M., & Choi, Y. (2013). Where not to eat? Improving public policy by predicting hygiene inspections using online reviews. Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (pp. 1443-1448).

Kannan, V., Shapiro, M. A., & Bilgic, M. (2019). Hindsight analysis of the Chicago food inspection forecasting model. arXiv.org.

Mahmood, A., & Khan, H. U. (2019). Identification of critical factors for assessing the quality of restaurants using data mining approaches. Electronic Library, 37(6), 952–969. https://doi.org/10.1108/EL-12-2018-0241

Mejia, J., Mankad, S., & Gopal, A. (2019). A for effort? Using the crowd to identify moral hazard in New York City restaurant hygiene inspections. Information Systems Research, 30(4), 1363–1386. https://doi.org/10.1287/isre.2019.0866

Shafieizadeh, K., Alotaibi, S., & Tao, C.-W.. (2023). Information processing of food safety messages: what really matters for restaurant customers? International Journal of Contemporary Hospitality Management, 35(10), 3638–3661. https://doi.org/10.1108/IJCHM-05-2022-0670

Schomberg, J. P., Haimson, O. L., Hayes, G. R., & Anton-Culver, H. (2016). Supplementing public health inspection via social media. PLOS One, 11(3), e0152117–e0152117. https://doi.org/10.1371/journal.pone.0152117

Wang, Z., Balasubramani, B. & Cruz, I (2017). Predictive analytics using text classification for restaurant inspections. Proceedings of the 3rd ACM SIGSPATIAL Workshop on Smart Cities and Urban Analytics. https://doi.org/10.1145/3152178.3152192

(9) Appendix

Appendix A – API Backend (Shlok)

chicago_fetcher.py

```
import requests


def fetch_chicago_violations(limit=5):
    url = f"https://data.cityofchicago.org/resource/4ijn-s7e5.json?$limit={limit}"
    response = requests.get(url)
    if response.status_code == 200:
        return response.json()
    else:
        return []
```

```
def search_chicago_violations(query, limit=10):
    url = f"https://data.cityofchicago.org/resource/4ijn-
s7e5.json?$q={query}&$limit={limit}"
    response = requests.get(url)
    if response.status_code == 200:
        return response.json()
    else:
        return []
```

preprocessing.py

```
def preprocess_violation(record):
    # Adjust based on the Chicago API fields
    violation_description = record.get("violation_description", "")
    facility_name = record.get("dba_name", "Unknown Restaurant")
    inspection_date = record.get("inspection_date", "Unknown Date")

    text = f"Violation at {facility_name} on {inspection_date}: {violation_description}"
    return text
```

model_loader.py

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from peft import PeftModel, PeftConfig

def load_model_and_tokenizer(model_path):
    # First load the base flan-t5-small model
    base_model = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-small")
```

```
    tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-small")


    # Then load the LoRA adapter on top
    model = PeftModel.from_pretrained(base_model, model_path)
    return model, tokenizer
```

inference.py

```
def summarize_violation(text, model, tokenizer, max_length=128):
    inputs = tokenizer(text, return_tensors="pt", truncation=True)
    outputs = model.generate(**inputs, max_new_tokens=max_length)
    summary = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return summary
```

app.py

```
from flask import Flask, request, jsonify, render_template
from model_loader import load_model_and_tokenizer
from inference import summarize_violation
from chicago_fetcher import fetch_chicago_violations, search_chicago_violations
from preprocessing import preprocess_violation


app = Flask(__name__)


# Load model once when the server starts
MODEL_PATH = "D:/T An/flan_t5_small_plain_lora"  # Change this path if needed
model, tokenizer = load_model_and_tokenizer(MODEL_PATH)


@app.route("/summarize_violation", methods=["POST"])
def summarize_violation_api():
    data = request.json
    text = data.get("text", "")
```

```
    if not text:
        return jsonify({"error": "No text provided"}), 400

    summary = summarize_violation(text, model, tokenizer)
    return jsonify({"summary": summary})

@app.route("/fetch_and_summarize", methods=["GET"])
def fetch_and_summarize_api():
    records = fetch_chicago_violations()
    results = []

    for record in records:
        raw_text = preprocess_violation(record)
        summary = summarize_violation(raw_text, model, tokenizer)
        results.append({
            "original": raw_text,
            "summary": summary
        })

    return jsonify(results)

@app.route("/search_and_summarize", methods=["POST"])
def search_and_summarize_api():
    data = request.json
    query = data.get("query", "")
    if not query:
        return jsonify({"error": "No query provided"}), 400

    records = search_chicago_violations(query)
```

```
    if not records:
        return jsonify({"error": "No violations found for the given search criteria."}),
404

    results = []

    for record in records:
        raw_text = preprocess_violation(record)
        summary = summarize_violation(raw_text, model, tokenizer)
        results.append({
            "original": raw_text,
            "summary": summary
        })

    return jsonify(results)

@app.route("/")
def home():
    return "<h2>Restaurant Violations Summarizer is Running!</h2>"

if __name__ == "__main__":
    app.run(debug=True)
```

Custom API Structure JSON

```
{
      "info": {
            "_postman_id": "8f15b531-380a-4ff9-9b42-ce6ff02ca50e",
            "name": "Restaurant Violations API",
```

```
            "schema":
"https://schema.getpostman.com/json/collection/v2.1.0/collection.json",
            "_exporter_id": "23664406"
      },
      "item": [
            {
                  "name": "Summarize Custom Violation",
                  "request": {
                        "method": "POST",
                        "header": [
                              {
                                    "key": "Content-Type",
                                    "value": "application/json"
                              }
                        ],
                        "body": {
                              "mode": "raw",
                              "raw": "{\n    \"text\": \"Rodent droppings found in food
preparation area.\"\n}"
                        },
                        "url": {
                              "raw": "http://127.0.0.1:5000/summarize_violation",
                              "protocol": "http",
                              "host": [
                                    "127",
                                    "0",
                                    "0",
                                    "1"
                              ],
```

```
                        "port": "5000",
                        "path": [
                              "summarize_violation"
                        ]
                  }
            },
            "response": []
      },
      {
            "name": "Fetch and Summarize Violations",
            "request": {
                  "method": "GET",
                  "header": [],
                  "url": {
                        "raw": "http://127.0.0.1:5000/fetch_and_summarize",
                        "protocol": "http",
                        "host": [
                              "127",
                              "0",
                              "0",
                              "1"
                        ],
                        "port": "5000",
                        "path": [
                              "fetch_and_summarize"
                        ]
                  }
            },
            "response": []
```

```
},
{
	"name": "Search and Summarize Violations",
	"request": {
		"method": "POST",
		"header": [
			{
				"key": "Content-Type",
				"value": "application/json"
			}
		],
		"body": {
			"mode": "raw",
			"raw": "{\n    \"query\": \"The Dearborn\"\n}"
		},
		"url": {
			"raw": "http://127.0.0.1:5000/search_and_summarize",
			"protocol": "http",
			"host": [
				"127",
				"0",
				"0",
				"1"
			],
			"port": "5000",
			"path": [
				"search_and_summarize"
			]
		}
```

```
                },
                "response": []
            }
        ]
}
```

Appendix B – Data Preparation & Modeling (Zahid)

Chicago_Health_Inspections.py

```python
# -*- coding: utf-8 -*-
"""Chicago Health Inspections.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1jAeF8u_gFYpPdRdKaoaYkC8xETlSC3Av

# Data Loading
"""

!pip install sodapy pandas
from sodapy import Socrata
import pandas as pd

APP_TOKEN = "8GXNVY2yK4u55lcft3CNx0PWO"    # replace with the token you got from Socrata
client = Socrata("data.cityofchicago.org", APP_TOKEN)

# grab a tiny sample to prove it works
results = client.get(
  "4ijn-s7e5",
  select="dba_name,address,inspection_date,results,violations",
  limit=5
)
df = pd.DataFrame.from_records(results)
df.head()

def search_inspections(name_query: str, limit: int = 50) -> pd.DataFrame:
    where_clause = f"upper(dba_name) LIKE '%{name_query.upper()}%'"
    results = client.get(
        "4ijn-s7e5",
        where=where_clause,
```

```
        select="dba_name,address,inspection_date,results,violations",
        order="inspection_date DESC",
        limit=limit
    )
    df = pd.DataFrame.from_records(results)
    df["inspection_date"] = pd.to_datetime(df["inspection_date"])
    return df

# Example
df_taco = search_inspections("TACO", limit=10)
df_taco

def search_inspections(name_query: str, limit: int = 50) -> pd.DataFrame:
    select_str = ",".join([
        "dba_name", "facility_type", "address", "city", "state", "zip",
        "inspection_date", "inspection_type", "risk", "results", "violations"
    ])
    where_clause = f"upper(dba_name) LIKE '%{name_query.upper()}%'"
    results = client.get(
        "4ijn-s7e5",
        where=where_clause,
        select=select_str,
        order="inspection_date DESC",
        limit=limit
    )
    df = pd.DataFrame.from_records(results)
    df["inspection_date"] = pd.to_datetime(df["inspection_date"])
    df["Address"] = (
        df["address"]
        .str.cat(df["city"], sep=", ")
        .str.cat(df["state"], sep=", ")
        .str.cat(df["zip"], sep=" ")
    )
    df = df.drop(columns=["address", "city", "state", "zip"])
    df = df.rename(columns={
        "dba_name":       "Business Name",
        "facility_type":  "Facility Type",
        "inspection_date":"Inspection Date",
        "inspection_type":"Inspection Type",
        "risk":           "Risk Category",
        "results":        "Result",
        "violations":     "Raw Violations"
    })
    return df
```

```python
# Try it:
df_sample = search_inspections("PIZZA", limit=10)
df_sample

def format_and_reorder(df: pd.DataFrame, drop_risk: bool = False) -> pd.DataFrame:
    df["Inspection Date"] = df["Inspection Date"].dt.strftime("%m/%d/%Y")
    if not drop_risk:
        df["Risk Level"] = df["Risk Category"].str.extract(r"\((.*?)\)")
    cols = [
        "Business Name", "Facility Type", "Address",
        "Inspection Date", "Inspection Type", "Raw Violations"
    ]
    if not drop_risk:
        cols.append("Risk Level")
    cols.append("Result")
    return df[cols]

def clean_and_reorder(df: pd.DataFrame) -> pd.DataFrame:
    df["Inspection Frequency"] = df["Risk Category"].str.extract(r"\((.*?)\)")
    df["Inspection Date"] = df["Inspection Date"].dt.strftime("%m/%d/%Y")
    df = df.drop(columns=["Risk Category"])
    cols = [
        "Business Name", "Facility Type", "Address",
        "Inspection Date", "Inspection Type",
        "Inspection Frequency", "Raw Violations", "Result"
    ]
    return df[cols]

inspection_type_map = {
    "Canvass":                "Routine Inspection",
    "Consultation":           "Pre-Opening Consultation",
    "Complaint":              "Complaint-Driven Inspection",
    "License":                "Licensing Inspection",
    "Suspect Food Poisoning": "Food-Poisoning Investigation",
    "Task-Force Inspection":  "Bar/Tavern Task-Force Inspection"
}

def humanize_inspection_type(df: pd.DataFrame) -> pd.DataFrame:
    def map_type(raw):
        base = raw.replace("Re-","")
        label = inspection_type_map.get(base, base)
        if raw.startswith("Re-"):
            return f"{label} (Re-Check)"
```

```
        return label

    df["Inspection Type"] = df["Inspection Type"].apply(map_type)
    return df

# Pipeline example:
df = search_inspections("PIZZA", limit=10)
df = clean_and_reorder(df)
df = humanize_inspection_type(df)
df

import pandas as pd

# 1. Ensure no NaNs in the source column
df["Raw Violations"] = df["Raw Violations"].fillna("")

# 2. Updated extractor
def extract_key_phrases(text, top_n: int = 5) -> str:
    if not isinstance(text, str) or not text.strip():
        return ""
    doc = nlp(text)
    seen, phrases = set(), []
    for chunk in doc.noun_chunks:
        tok = chunk.text.strip().lower()
        if tok not in seen:
            seen.add(tok)
            phrases.append(chunk.text)
        if len(phrases) >= top_n:
            break
    return ", ".join(phrases)

# 3. Re-run the pipeline
df = search_inspections("PIZZA", limit=10)
df = clean_and_reorder(df)
df = humanize_inspection_type(df)
df["Key Phrases"] = df["Raw Violations"].apply(extract_key_phrases)
df
```

Creating Training Splits for Transformer (Health Inspections).py

```
# -*- coding: utf-8 -*-
"""Creating Training Splits for Transformer (Health Inspections).ipynb
```

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1UOgl6sTNB64B6TA0K_lJg30IIzg16np5

📄 Executive Summary: Health Violations Dataset Enrichment Pipeline

Overview
This project builds a **high-quality**, **augmented**, and **structured** dataset of restaurant health inspection violations, preparing it for downstream NLP modeling.

Steps

1. Data Sampling and Splitting
- Loaded the full `Food_Inspections` dataset.
- Filtered out empty "Violations" entries.
- Randomly sampled **3,000** examples.
- Split into:
  - **Train** (80%)
  - **Validation** (10%)
  - **Test** (10%)
- Saved each split as CSV files.

2. Training Set Enrichment
- Ran **parallel GPT API calls** to:
  - Generate **plain-language summaries**.
  - Assign **3–7 customer-friendly keywords**.
  - Enforce a **standardized safety verdict** at the end of each summary.
- Implemented **retry logic**, **backoff**, and **rate limit handling**.
- Dropped any samples that failed parsing or exceeded retry attempts.

3. Synthetic Data Augmentation
- Generated **100 fictional** "unsafe to eat here" violations to address class imbalance.
- Enriched synthetic violations with summaries and keywords using the same GPT pipeline.
- Saved synthetic examples separately.

4. Dataset Merging
- Merged real and synthetic examples into a final training set
(`violations_train_final.jsonl`).
- Shuffled the merged dataset to prevent ordering bias.

```
### 5. Validation and Test Enrichment
- Enriched the validation and test splits using the same pipeline.
- Saved results into `violations_val.jsonl` and `violations_test.jsonl`.

---

## Quality Checks
- Verified that each summary ends with exactly one of the predefined verdicts.
- Random sampling of outputs to manually inspect data quality.
- (Note: Summary verdict distribution was planned but minor missing code at the end.)

---

## Outputs
- `violations_train_final.jsonl` → Final train set (real + synthetic).
- `violations_val.jsonl` → Validation set.
- `violations_test.jsonl` → Test set.
- All saved in clean JSONL format for seamless ingestion.

---

# Training Set
"""

!pip install openai --quiet

# ── 0. Constants & Imports ──────────────────────────────────────────
import pandas as pd
import time
from concurrent.futures import ThreadPoolExecutor, as_completed
from tqdm.auto import tqdm
from openai import OpenAI, RateLimitError
from math import ceil
import json, re
from collections import Counter

# adjust to your org's limits
MAX_RETRIES   = 5
TARGET_RPM    = 300           # your total allowed calls/minute
MAX_WORKERS   = 5             # number of threads you actually want to run
SLEEP_PER_CALL = (60 / TARGET_RPM) * MAX_WORKERS  # seconds each thread should wait after a
successful call
```

```python
# Instantiate the OpenAI client (paste your key below)
client = OpenAI(api_key="Redacted")

# ── 1. Load & Split ──────────────────────────────────────────────

# 1. Load the full violations dataset
df = pd.read_csv('/content/Food_Inspections_20250419.csv')

# 2. Filter out any missing "Violations"
df = df.dropna(subset=['Violations'])

# 3. Sample 3,000 rows for processing
df_sample = df.sample(n=3000, random_state=42)[['Violations']]

# 4. Split into train/validation/test (80/10/10)
train_df = df_sample.sample(frac=0.8, random_state=42)
rem_df   = df_sample.drop(train_df.index)
val_df   = rem_df.sample(frac=0.5, random_state=42)
test_df  = rem_df.drop(val_df.index)

# 5. Save each split to CSV
train_df.to_csv('violations_train.csv', index=False)
val_df.to_csv('violations_val.csv', index=False)
test_df.to_csv('violations_test.csv', index=False)

# Display confirmation
print("Splits saved: ")
print(f"  train: {len(train_df)} rows")
print(f"  val:   {len(val_df)} rows")
print(f"  test:  {len(test_df)} rows")

# ── 2. Prep for Parallel Processing ──────────────────────────────

# 1. Load the training split you just created
df = pd.read_csv("/content/violations_train.csv")

# 2. Ensure 'Violations' is a non-empty string column
df["Violations"] = df["Violations"].astype(str).str.strip()
df = df[df["Violations"] != ""]

# 3. Prepare for parallel processing
total = len(df)
```

```python
outputs = [None] * total

print(f"Ready to process {total} violations.")

# ─── 3. Worker fn: retry + backoff + throttle ──────────────────────────

def process_violation(args):
    idx, vio = args
    prompt = (
    "You are given one or more violation descriptions from a health inspection report. Each violation may or may not be related to the safety of the food being served.\n"
    "Your task is to generate a valid JSON object with exactly two keys:\n"
    '  "summary": A plain-language 1–2 sentence explanation for customers. Focus on whether any of the violations suggest a risk to food safety. Your summary should end with exactly one of the following:\n'
    '    - "This violation is not related to food safety."\n'
    '    - "This violation is related to food safety, but it is safe to eat here."\n'
    '    - "This violation is related to food safety and may make it unsafe to eat here."\n'
    '    - "This violation is related to food safety and it is not safe to eat here."\n'
    '  "keywords": An array of 3–7 lowercase, customer-friendly keywords that summarize the key issues (e.g., "sanitation", "pests", "utensils", "equipment", "food contact").\n'
    "Do not include the raw violation text or any additional commentary.\n"
    f"Violations: {vio}"
    )
    backoff = SLEEP_PER_CALL

    for attempt in range(1, MAX_RETRIES + 1):
        try:
            resp = client.chat.completions.create(
                model="gpt-4o-mini-2024-07-18",
                messages=[{"role": "user", "content": prompt}],
                temperature=0.2,
                max_tokens=300,
            )
            return idx, resp.choices[0].message.content.strip()

        except RateLimitError as e:
            headers = getattr(e, "headers", {}) or {}
            retry_after = float(headers.get("Retry-After", backoff))
            time.sleep(max(1, ceil(retry_after)))
            backoff = min(backoff * 2, 60)
        except Exception as e:
            if attempt == MAX_RETRIES:
                print(f"❌ Fatal error on idx={idx}: {e}")
```

```
                return idx, "[ERROR]"
            time.sleep(1)

# ── 4. Run in ThreadPool & Collect ──────────────────────────────
#Took 10 minutes

# Submit all violations to be processed in parallel
with ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
    futures = {
        executor.submit(process_violation, (i, v)): i
        for i, v in enumerate(df["Violations"])
    }

    for future in tqdm(as_completed(futures),
                       total=total,
                       desc="Processing rows",
                       unit="row"):
        idx, result = future.result()
        outputs[idx] = result

# ── 5. Drop errors & write JSONL ────────────────────────────────

# Filter out any failures
clean = [(vio, r) for vio, r in zip(df["Violations"], outputs) if r != "[ERROR]"]
print(f"Dropped {len(df) - len(clean)} error rows; writing {len(clean)} clean JSON objects.")

# Write out the clean train split as JSONL
with open("violations_train.jsonl", "w") as outf:
    for vio, resp in clean:
        try:
            parsed = json.loads(resp)
        except json.JSONDecodeError:
            continue
        clean_obj = {
            "violation": vio,
            "summary":   parsed.get("summary"),
            "keywords":  parsed.get("keywords")
        }
        outf.write(json.dumps(clean_obj) + "\n")
print("☑ Wrote violations_train.jsonl")

# ── 1.7 Generate & Enrich Synthetic “Not Safe” Violations ──────────
```

```python
def generate_synthetic_violations(n_samples=100, batch_size=5):
    synthetic_violations = []
    total_batches = n_samples // batch_size
    for batch_idx in tqdm(range(total_batches), desc="Generating synthetic violations",
unit="batch"):
        prompt = (
            "You are creating synthetic restaurant health inspection violations that would
clearly "
            "result in the verdict: \"This violation is related to food safety and it is not
safe to eat here.\"\n"
            f"Generate {batch_size} fictional violations. Each one must:\n"
            "- Start with a realistic inspector-style comment (no need for code numbers or
titles).\n"
            "- Describe a serious food-safety hazard (e.g., raw meat at room temp, sewage,
pests).\n"
            "- Use inspection tone (words like 'observed', 'must', 'discarded').\n"
            "- Output each violation separated by two newlines, with no summaries or JSON.\n"
        )
        try:
            resp = client.chat.completions.create(
                model="gpt-4o-mini-2024-07-18",
                messages=[{"role":"user","content":prompt}],
                temperature=0.8,
                max_tokens=200,
            )
            batch = [v.strip() for v in resp.choices[0].message.content.split("\n\n") if
v.strip()]
            synthetic_violations.extend(batch)
        except Exception as e:
            print(f"❌ Error generating batch {batch_idx}: {e}")
    print(f"☑ Generated {len(synthetic_violations)} synthetic violations.")
    return synthetic_violations[:n_samples]


def enrich_synthetic_violations(violation_list, output_jsonl="violations_synthetic.jsonl"):
    total = len(violation_list)
    outputs = [None]*total

    def worker(args):
        idx, vio = args
        prompt = (
              "You are given a violation description. Output a JSON object with exactly two
keys:\n"
              '  "summary": 1–2 sentence consumer-friendly verdict ending with '
```

```
            '"This violation is related to food safety and it is not safe to eat here."\n'
            '  "keywords": array of 3–7 lowercase keywords summarizing key issues.\n'
            "Do not include the raw violation text or any additional commentary.\n"
            f"Violation: {vio}"
        )
    backoff = SLEEP_PER_CALL
    for attempt in range(1, MAX_RETRIES+1):
        try:
            r = client.chat.completions.create(
                model="gpt-4o-mini-2024-07-18",
                messages=[{"role":"user","content":prompt}],
                temperature=0.2, max_tokens=200
            )
            return idx, r.choices[0].message.content.strip()
        except RateLimitError as e:
            retry = float(getattr(e, "headers",{}).get("Retry-After", backoff))
            time.sleep(max(1, retry))
            backoff = min(backoff*2,60)
        except:
            if attempt==MAX_RETRIES:
                return idx, "[ERROR]"
            time.sleep(1)

with ThreadPoolExecutor(max_workers=MAX_WORKERS) as ex:
    futures = { ex.submit(worker,(i,v)):i
                for i,v in enumerate(violation_list) }
    for future in tqdm(as_completed(futures), total=total, desc="Enriching synthetic"):
        i, resp = future.result()
        outputs[i]=resp

clean = [(v,r) for v,r in zip(violation_list, outputs) if r!="[ERROR]"]
with open(output_jsonl, "w") as outf:
    for vio, resp in clean:
        try:
            parsed = json.loads(resp)
        except json.JSONDecodeError:
            continue
        clean_obj = {
            "violation": vio,
            "summary":   parsed.get("summary"),
            "keywords":  parsed.get("keywords")
        }
        outf.write(json.dumps(clean_obj) + "\n")
print(f"☑ Wrote {output_jsonl} ({len(clean)} rows)")
```

```python
def merge_jsonl_files(real_jsonl="violations_train.jsonl",
                      synthetic_jsonl="violations_synthetic.jsonl",
                      output_jsonl="violations_train_final.jsonl"):
    with open(output_jsonl,"w") as out:
        for fname in (real_jsonl, synthetic_jsonl):
            with open(fname) as inp:
                for line in inp:
                    out.write(line)
    print(f"☑ Merged into {output_jsonl}")

# ── 1.8 Run Synthetic Augmentation ──────────────────────────────

# 1) Generate 100–200 “not safe” violation comments
synthetic = generate_synthetic_violations(n_samples=100)

# 2) Enrich them to JSONL
enrich_synthetic_violations(synthetic, "violations_synthetic.jsonl")

# 3) Merge real+synthetic into final train file
merge_jsonl_files(
    real_jsonl="violations_train.jsonl",
    synthetic_jsonl="violations_synthetic.jsonl",
    output_jsonl="violations_train_final.jsonl"
)

#Print the first 5 of the train.json
!head -n 5 violations_train_final.jsonl

pattern = re.compile(
    r'(This violation is not related to food safety\.|'
    r'This violation is related to food safety, but it is safe to eat here\.|'
    r'This violation is related to food safety and may make it unsafe to eat here\.|'
    r'This violation is related to food safety and it is not safe to eat here\.)\s*$'
)

with open("violations_train_final.jsonl") as f:
    for i, line in enumerate(f):
        summary = json.loads(line)["summary"]
        if not pattern.search(summary):
            print(f"Line {i}: {summary}")
```

```
# ── 1.9 Optional: Shuffle the final training examples ──────────────
import random
lines = open("violations_train_final.jsonl").read().splitlines()
random.shuffle(lines)
with open("violations_train_final.jsonl","w") as f:
    f.write("\n".join(lines))
print("☑ Shuffled final train file")

# Preview first few synthetic enrichments
with open("violations_synthetic.jsonl") as f:
    for _ in range(5):
        print(json.loads(f.readline()), "\n---")

print("\n📊 Final Summary Verdict Distribution:")
for k, v in counts.items():
    print(f"{k}: {v}")

"""# Validation"""

def enrich_to_jsonl(input_csv: str, output_jsonl: str):
    # 1. Load & clean
    df = pd.read_csv(input_csv)
    df["Violations"] = df["Violations"].astype(str).str.strip()
    df = df[df["Violations"] != ""]

    total = len(df)
    outputs = [None] * total

    # 2. Parallel GPT calls
    with ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
        futures = {
            executor.submit(process_violation, (i, v)): i
            for i, v in enumerate(df["Violations"])
        }
        for future in tqdm(as_completed(futures),
                           total=total,
                           desc=f"Processing {input_csv}",
                           unit="row"):
            idx, result = future.result()
            outputs[idx] = result

    # 3. Filter errors
    clean = [(vio, r) for vio, r in zip(df["Violations"], outputs) if r != "[ERROR]"]
```

```
    with open(output_jsonl, "w") as outf:
        for vio, resp in clean:
            try:
                parsed = json.loads(resp)
            except json.JSONDecodeError:
                continue
            clean_obj = {
                "violation": vio,
                "summary":   parsed.get("summary"),
                "keywords":  parsed.get("keywords")
            }
            outf.write(json.dumps(clean_obj) + "\n")
    print(f"☑ Wrote {output_jsonl} ({len(clean)} rows)")

enrich_to_jsonl("violations_val.csv", "violations_val.jsonl")

"""# Test"""

enrich_to_jsonl("violations_test.csv", "violations_test.jsonl")
```

Fine-tuning transformer (FLAN-T5) & Evaluation NLP_Group_Project.py

This code is repetitive. I do the same thing essentially twice.

```
# -*- coding: utf-8 -*-
"""Fine-tuning transformer(FLAN-T5) & Evaluation) |NLP_Group Project .ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1C5iE1K23OADkEbN05WaLjxm4jKVk_Tm3

# Executive Summary

- **Project Goal**
  Build a local, quantized Transformer that ingests raw health-inspection violations and
returns a structured JSON object with two fields:
  1. **summary**: a 1–2 sentence consumer-friendly verdict ending in one of four canned food-
safety statements
  2. **keywords**: a list of 3–7 lowercase issue tags

- **Data Preparation (Notebook1- Not this one)**
  - Sampled 3,000 violation texts from Chicago’s API
```

```
  - Used OpenAI ChatCompletion to generate training JSONL (`summary` + `keywords`)
  - Split into train/val/test and augmented with synthetic “not safe” examples

- **Fine-Tuning (Notebook 2-This notebook)**
  - Loaded `google/flan-t5-small` with 4-bit quantization and applied LoRA adapters
  - Trained for 3 epochs on the JSONL data
  - Saved the quantized + LoRA model for offline inference

- **Evaluation Attempts**
  - **Plain-text metrics**: swapped to a “Summarize this…” prompt and computed ROUGE/BERTScore; saw small improvements (e.g. ROUGE-1 from ~0.13→0.15) but lost JSON structure
  - **JSON-parsing metrics**: tried to parse model outputs back into JSON and score summaries/keywords; scores stayed at zero because the inference prompt didn’t match training
  - **Debugging**: re-loaded models correctly, batched inference with progress bars, handled malformed outputs—nothing overcame the prompt-mismatch issue

- **Key Insight**
  The fine-tuned model only emits valid JSON when given the exact same instruction it saw during training. Any deviation (e.g. plain summarization prompt) collapses to empty or gibberish outputs, making robust metric computation impossible without re-training.

- **Next Steps: Web Integration**
  - Deploy the quantized + LoRA model in the backend
  - At inference, prepend the original JSON-generation prompt to each violation text
  - Parse `summary` + `keywords` from the returned JSON and serve to the frontend
  - Defer deeper offline metric refinement until after integration, since end-to-end functionality is now validated and cost-free (no API calls)

# Refined Code (1st Run)
"""

# Block A: Imports & Environment Setup

# Install necessary libraries
!pip install transformers datasets peft bitsandbytes accelerate evaluate -q

# Disable Weights & Biases logging
import os
os.environ["WANDB_DISABLED"] = "true"

# Core imports
import torch
```

```python
import pandas as pd
from datasets import load_dataset, DatasetDict
from transformers import (
    AutoTokenizer,
    T5ForConditionalGeneration,
    BitsAndBytesConfig,
    Seq2SeqTrainingArguments,
    Seq2SeqTrainer,
    DataCollatorForSeq2Seq
)
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
from tqdm.auto import tqdm
import evaluate
import json

# Check environment
print(f"PyTorch version: {torch.__version__}")

from datasets import load_dataset

data_files = {
    "train": "violations_train_final.jsonl",
    "validation": "violations_val.jsonl",
    "test": "violations_test.jsonl"
}

data = load_dataset("json", data_files=data_files)

# Inspect the dataset
print(data)
print("\nSample training example:")
print(data["train"][0])

# 2. Tokenizer
model_name = "google/flan-t5-small"
tokenizer = AutoTokenizer.from_pretrained(model_name)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

# 3. Preprocessing Function
max_input_length = 512
max_target_length = 128
```

```
def preprocess_fn(examples):
    inputs = tokenizer(
        examples["violation"],
        max_length=max_input_length,
        truncation=True,
        padding="max_length"
    )
    targets = [
        json.dumps({"summary": s, "keywords": k})
        for s, k in zip(examples["summary"], examples["keywords"])
    ]
    with tokenizer.as_target_tokenizer():
        tokenized_targets = tokenizer(
            targets,
            max_length=max_target_length,
            truncation=True,
            padding="max_length"
        )
    # mask pad tokens as -100
    labels = [
        [(tok if tok != tokenizer.pad_token_id else -100) for tok in seq]
        for seq in tokenized_targets["input_ids"]
    ]
    inputs["labels"] = labels
    return inputs

# 4. Apply Tokenization
tokenized = data.map(
    preprocess_fn,
    batched=True,
    remove_columns=data["train"].column_names
)

# 5. Quantization + LoRA Setup
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16  # A100-friendly
)
base_model = T5ForConditionalGeneration.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True
)
```

```
model = prepare_model_for_kbit_training(base_model)

lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="SEQ_2_SEQ_LM"
)
model = get_peft_model(model, lora_config)

# 6. Data Collator
data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)

# 7. Training Arguments (use BF16, disable FP16)
training_args = Seq2SeqTrainingArguments(
    output_dir="flan_t5_small_lora",
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    gradient_accumulation_steps=1,
    eval_strategy="epoch",
    save_strategy="epoch",
    logging_strategy="steps",
    logging_steps=50,
    num_train_epochs=3,
    learning_rate=2e-5,
    bf16=True,
    fp16=False,
    push_to_hub=False,
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    save_total_limit=2
)

# 8. Sanity Check: single-example forward-pass
batch = data_collator([ tokenized["train"][0] ])
batch = {k: v.to(next(model.parameters()).device) for k, v in batch.items()}
out = model(**batch)
print("sanity loss:", out.loss)  # should be finite (e.g. ~3–5)

# 9. Trainer Instantiation
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
```

```
    train_dataset=tokenized["train"],
    eval_dataset=tokenized["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer
)

# 10. Training & Evaluation
trainer.train()
print("Validation metrics:", trainer.evaluate(tokenized["validation"]))
print("Test metrics:",       trainer.evaluate(tokenized["test"]))

# ── Quick Evaluation: Plain-Text Summaries Only ──────────────────────

# 1. Install & imports (if not already done)
!pip install rouge_score evaluate bert_score --quiet

import pandas as pd
import evaluate
from transformers import T5ForConditionalGeneration

# 2. Load metrics
rouge = evaluate.load("rouge")
berts = evaluate.load("bertscore")

# 3. Prepare test data
violations = data["test"]["violation"]
references = data["test"]["summary"]   # your 1–2 sentence refs

# 4. Generation helper (plain summaries)
def generate_summaries(model, tokenizer, texts,
                       batch_size=16, max_new=128, min_len=20):
    model.eval()
    outs = []
    for i in range(0, len(texts), batch_size):
        chunk = texts[i : i + batch_size]
        enc = tokenizer(
            ["Summarize this inspection violation:\n\n" + t for t in chunk],
            return_tensors="pt",
            truncation=True,
            padding=True
        ).to(model.device)
        gen = model.generate(
            **enc,
            max_new_tokens=max_new,
```

```
            min_length=min_len,
            num_beams=4,
            early_stopping=True
        )
        decs = tokenizer.batch_decode(gen, skip_special_tokens=True)
        outs.extend([d.strip() for d in decs])
    return outs

# 5. Load your two models
base_model = T5ForConditionalGeneration.from_pretrained("google/flan-t5-
small").to(model.device)
ft_model   = model   # the LoRA-fine-tuned model you just trained

# 6. Generate summaries
print("Generating with base model…")
base_summaries = generate_summaries(base_model, tokenizer, violations)
print("Generating with fine-tuned model…")
ft_summaries   = generate_summaries(ft_model,   tokenizer, violations)

# 7. Sanity-check a few
for i in range(3):
    print(f"[{i}] BASE :", repr(base_summaries[i]))
    print(f"[{i}] F-T  :", repr(ft_summaries[i]))
    print(f"[{i}] REF  :", repr(references[i]))
    print("---")

# 8. Compute metrics
r_base = rouge.compute(predictions=base_summaries, references=references)
r_ft   = rouge.compute(predictions=ft_summaries,   references=references)
b_base = bermodel = berts.compute(predictions=base_summaries,
                                  references=references,
                                  model_type="microsoft/deberta-xlarge-mnli")
b_ft   = berts.compute(predictions=ft_summaries,
                       references=references,
                       model_type="microsoft/deberta-xlarge-mnli")

# 9. Aggregate & show
df = pd.DataFrame([
    {
      "model":"Base",
      "rouge1": r_base["rouge1"],
      "rouge2": r_base["rouge2"],
      "rougeL": r_base["rougeL"],
      "bertscore": sum(b_base["f1"])/len(b_base["f1"])
```

```
    },
    {
      "model":"Fine-tuned",
      "rouge1": r_ft["rouge1"],
      "rouge2": r_ft["rouge2"],
      "rougeL": r_ft["rougeL"],
      "bertscore": sum(b_ft["f1"])/len(b_ft["f1"])
    }
]).set_index("model")

from IPython.display import display
display(df)

# 1. Save the fine-tuned model & tokenizer
model.save_pretrained("flan_t5_small_plain_lora")
tokenizer.save_pretrained("flan_t5_small_plain_lora")

# 2. (Optional) zip the folder for easy download/storage
!zip -r flan_t5_small_plain_lora.zip flan_t5_small_plain_lora

"""# Refined Code (Loading Model in-2nd Run)

The Core Issue
Train-time inputs: "<RAW_VIOLATION_TEXT>"

Train-time target: {"summary":…, "keywords":[…]}

Eval-time inputs: "<RAW_VIOLATION_TEXT>" (no prefix)

Eval-time tried: also tried "Summarize this…" + text

In both cases it didn't see the big JSON-instruction at inference the way it did in training,
so it just loops or outputs nothing.
"""

# unzip your saved model
!unzip flan_t5_small_plain_lora.zip -d flan_t5_small_plain_lora

# Cell 1: Installs, imports, device
!pip install transformers datasets peft evaluate rouge_score bert_score -q

import torch, json
from transformers import BitsAndBytesConfig, T5ForConditionalGeneration, AutoTokenizer
```

```python
from peft import PeftModel
import evaluate, pandas as pd
from datasets import load_dataset

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

# Cell 2: Load test data
data = load_dataset("json", data_files={"test":"violations_test.jsonl"})
texts        = data["test"]["violation"]
ref_summ     = data["test"]["summary"]
ref_keywords = data["test"]["keywords"]

# Cell 3: Load quantized + LoRA model
bnb = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)
backbone  = T5ForConditionalGeneration.from_pretrained(
    "google/flan-t5-small",
    quantization_config=bnb,
    device_map="auto",
    trust_remote_code=True
)
model_dir = "/content/flan_t5_small_plain_lora/flan_t5_small_plain_lora"
model     = PeftModel.from_pretrained(backbone, model_dir).to(device)
tokenizer = AutoTokenizer.from_pretrained(model_dir)

# Cell 4 (updated): Inference (raw violation → JSON or list)
def infer_violation(vio: str):
    enc = tokenizer(vio, return_tensors="pt", truncation=True, padding=True).to(device)
    gen = model.generate(
        **enc,
        max_new_tokens=256,
        num_beams=4,
        early_stopping=True
    )
    raw = tokenizer.decode(gen[0], skip_special_tokens=True)
    try:
        obj = json.loads(raw)
    except json.JSONDecodeError:
        return {"summary": "", "keywords": []}

    # If they gave a dict, great
```

```
    if isinstance(obj, dict):
        return obj

    # If they gave a list [summary, keywords], unpack it
    if isinstance(obj, list) and len(obj) == 2:
        summary, keywords = obj
        return {
            "summary": summary if isinstance(summary, str) else "",
            "keywords": keywords if isinstance(keywords, list) else []
        }

    # Otherwise fallback
    return {"summary": "", "keywords": []}
# Cell 5: Run inference & collect predictions (with progress bar)
from tqdm.auto import tqdm

pred_summaries, pred_keywords = [], []
batch_size = 16

for i in tqdm(range(0, len(texts), batch_size), desc="Running inference"):
    batch = texts[i : i + batch_size]
    enc = tokenizer(batch, return_tensors="pt", truncation=True, padding=True).to(device)
    gens = model.generate(
        **enc,
        max_new_tokens=256,
        num_beams=4,
        early_stopping=True
    )
    raws = tokenizer.batch_decode(gens, skip_special_tokens=True)
    for raw in raws:
        try:
            parsed = json.loads(raw)
        except json.JSONDecodeError:
            # nothing valid → empty
            summary, keywords = "", []
        else:
            if isinstance(parsed, dict):
                summary  = parsed.get("summary", "")
                keywords = parsed.get("keywords", [])
            elif isinstance(parsed, list) and len(parsed) == 2:
                s, k = parsed
                summary  = s if isinstance(s, str) else ""
                keywords = k if isinstance(k, list) else []
            elif isinstance(parsed, str):
```

```
                # model returned a bare string
                summary, keywords = parsed, []
            else:
                summary, keywords = "", []

        pred_summaries.append(summary)
        pred_keywords.append(keywords)

# Cell 6: Compute and display metrics
# 6a) ROUGE on summaries
rouge        = evaluate.load("rouge")
rouge_scores = rouge.compute(predictions=pred_summaries, references=ref_summ)

# 6b) Set-based F1 for keywords
def keyword_f1(pred, ref):
    p, r = set(pred), set(ref)
    tp   = len(p & r)
    return 2*tp/(len(p)+len(r)) if (p and r) else 0.0

kw_f1s    = [keyword_f1(p, r) for p, r in zip(pred_keywords, ref_keywords)]
avg_kw_f1 = sum(kw_f1s) / len(kw_f1s)

# 6c) Show results
df = pd.DataFrame([{
    "rouge1": rouge_scores["rouge1"],
    "rouge2": rouge_scores["rouge2"],
    "rougeL": rouge_scores["rougeL"],
    "avg_keyword_f1": avg_kw_f1
}], index=["Fine-tuned"])

print(df)
```

Appendix C – Frontend UI (Carolina)

eat-safe-frontend/

```
├── public/
│   └── index.html         # The main HTML file loaded in the browser
├── src/
```

```
│   ├── api.js            # Handles talking to the backend or dummy data
│   ├── theme.js          # Sets up the look and feel (colors, fonts)
│   ├── App.js            # The main React component (the app itself)
│   ├── index.js          # Starts the app and puts it on the web page
│   └── index.css         # (Optional) Global CSS styles
├── package.json          # Lists dependencies and scripts
├── README.md             # Basic project info
```

public/index.html

```
<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="utf-8" />

    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />

    <meta name="viewport" content="width=device-width, initial-scale=1" />

    <meta name="theme-color" content="#000000" />

    <meta

      name="description"

      content="Eat Safe - Check restaurant safety information"

    />

    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />

    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />

    <title>Eat Safe App</title>

    <link rel="stylesheet"
href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap" />

    <link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons"
/>

  </head>

  <body>

    <noscript>You need to enable JavaScript to run this app.</noscript>

    <div id="root"></div>

  </body>
```

```
</html>
```

src/api.js

```
// Import the Axios library for making HTTP requests
import axios from 'axios';

// --- Configuration Constants ---

// API_URL: Backend endpoint that accepts a JSON body { query: "<search term>" }.
// IMPORTANT: Replace the placeholder URL with your production endpoint.
const API_URL = 'http://localhost:5000';

// USE_DUMMY_API: Toggle between simulated data and real HTTP requests.
const USE_DUMMY_API = false;

// --- Dummy API Simulation Function ---

/**
 * Simulates fetching data from an API endpoint (development only).
 *
 * @param {string} query - The search term entered by the user.
 * @returns {Promise<{ summary: string, keywords: string[] }>}
 */
const fetchDummyData = (query) => {
  console.log(`Simulating API call for query: ${query}`);

  return new Promise((resolve) => {
    setTimeout(() => {
      let summary = `This is a sample summary for "${query}". It provides safety
information based on mock data.`;
      let keywords = ['mock data', 'safe', 'restaurant', query.toLowerCase()];

      if (query.toLowerCase().includes('pizza')) {
        summary = `Detailed safety report for pizza places matching "${query}". Found
potential hygiene concerns in mock data.`;
        keywords = ['pizza', 'hygiene concern', 'mock data', 'warning'];
      } else if (query.toLowerCase().includes('taco')) {
        summary = `Safety analysis for taco vendors related to "${query}". Mock data
indicates good standing.`;
        keywords = ['taco', 'good standing', 'safe', 'mock data'];
      }
```

```
      resolve({ summary, keywords });
    }, 1500);
  });
};

// --- Main Data-Fetching Function ---

/**
 * Fetches restaurant-safety data for the provided query.
 *
 * @param {string} query - Restaurant name or address fragment.
 * @returns {Promise<{ summary: string, keywords: string[] }>}
 */
export const fetchSafetyData = async (query) => {
  if (USE_DUMMY_API) {
    // ----- Use Dummy Stub -----
    return fetchDummyData(query);
  }

  // ----- Use Real API (POST JSON) -----
  console.log(`Posting to real API: ${API_URL}`);

  try {
    const response = await axios.post(
      API_URL + "/api/restaurant-info",
      { query },                                  // request body
      { headers: { 'Content-Type': 'application/json' } }
    );

    // Basic shape validation
    if (
      response.data &&
      typeof response.data.summary === 'string' &&
      Array.isArray(response.data.keywords)
    ) {
      return response.data;
    } else {
      console.error('Invalid API response format:', response.data);
      throw new Error('Received invalid data format from the server.');
    }
  } catch (error) {
    console.error('Axios request failed:', error);
```

```
    if (error.response) {
      throw new Error(
        `API Error: ${error.response.status} - ${
          error.response.data?.message || error.response.statusText
        }`
      );
    } else if (error.request) {
      throw new Error('Network Error: No response received from server.');
    } else {
      throw new Error(`Request Setup Error: ${error.message}`);
    }
  }
};
```

src/App.js

```
// Import necessary React hooks and components
import React, { useState, useCallback } from 'react';
// Import the API fetching function and the custom theme
import { fetchSafetyData } from './api';
import theme from './theme';
// Import Material UI components for building the user interface
import {
  Container,        // Main layout container
  TextField,        // Input field for search query
  Button,           // Search button
  Box,              // Generic container component for layout and styling
  Typography,       // Text elements (headings, paragraphs)
  CircularProgress, // Loading indicator
  Alert,            // Component for displaying error messages
  Chip,             // Used to display keywords
  CssBaseline,      // Provides consistent baseline styling across browsers
  ThemeProvider,    // Applies the custom theme to descendant components
  InputAdornment    // Used to add icons inside the TextField
```

```
} from '@mui/material';

// Import the Search icon from Material UI icons

import SearchIcon from '@mui/icons-material/Search';

// Define the background image URL

// Replace with the actual Unsplash URL or a locally hosted image path

const backgroundImageUrl = 'https://images.unsplash.com/photo-1555396273-
367ea4eb4db5?q=80&w=1974&auto=format&fit=crop'; // Example: A different restaurant
interior

// --- Main Application Component ---

/**

 * The main functional component for the Eat Safe application.

 * It manages the application state (search query, results, loading status, errors)

 * and renders the user interface, including the search bar and results display area.

 */

function App() {

  // --- State Management using useState hook ---

  const [query, setQuery] = useState('');

  const [results, setResults] = useState(null);

  const [loading, setLoading] = useState(false);

  const [error, setError] = useState(null);

  // --- Event Handlers ---

  const handleInputChange = (event) => {

    setQuery(event.target.value);

    if (results || error) {

        setResults(null);

        setError(null);

    }

  };

  const handleSearch = useCallback(async () => {
```

```
  if (!query.trim()) {
    setError('Please enter a search query.');
    setResults(null);
    return;
  }
  setLoading(true);
  setError(null);
  setResults(null);
  try {
    const data = await fetchSafetyData(query);
    setResults(data);
  } catch (err) {
    console.error('Search failed:', err);
    setError(err.message || 'Failed to fetch data. Please try again.');
    setResults(null);
  } finally {
    setLoading(false);
  }
}, [query]);
const handleKeyPress = (event) => {
  if (event.key === 'Enter') {
    handleSearch();
  }
};
// --- Render Logic (JSX) ---
return (
  <ThemeProvider theme={theme}>
    <CssBaseline />
    {/* Outer Box to apply background image */}
```

```
      <Box sx={{
        minHeight: '100vh', // Ensure box takes at least full viewport height
        backgroundImage: `url(${backgroundImageUrl})`,
        backgroundSize: 'cover', // Cover the entire area
        backgroundPosition: 'center', // Center the image
        backgroundRepeat: 'no-repeat',
        display: 'flex', // Use flexbox to center content vertically
        alignItems: 'center', // Center content vertically
        py: 4, // Add some vertical padding
      }}>
        {/* Main content container */}
        {/* Removed maxWidth="md" to allow content to potentially stretch more */}
        {/* Added background color with opacity to make text readable over image */}
        <Container sx={{ backgroundColor: 'rgba(255, 255, 255, 0.1)', backdropFilter:
'blur(5px)', borderRadius: '16px', p: 4 }}>
          {/* Application Header - Adjusted color for better contrast */}
          <Typography variant="h4" component="h1" gutterBottom align="center" sx={{
color: 'white', textShadow: '1px 1px 3px rgba(0,0,0,0.7)', mb: 4 }}>
            Eat Safe: Restaurant Safety Check for Chicago
          </Typography>
          {/* Search Bar Area - Styled like the reference image */}
          <Box
            component="form"
            onSubmit={(e) => { e.preventDefault(); handleSearch(); }}
            sx={{
              display: 'flex',
              alignItems: 'center',
              gap: 1,
              p: 3, // Padding inside the box
```

```
              mb: 4,
              width: '100%',
              maxWidth: '600px',
              mx: 'auto',
              backgroundColor: 'white', // White background for the search box
              borderRadius: '16px', // Modern rounded corners
              boxShadow: 3, // Subtle shadow for depth (theme shadow)
            }}
            noValidate
            autoComplete="off"
          >
            {/* Search Input Field */}
            <TextField
              fullWidth
              variant="outlined" // Use outlined for better look on white bg
              label="Search Restaurant Name or Address"
              value={query}
              onChange={handleInputChange}
              onKeyPress={handleKeyPress}
              disabled={loading}
              InputProps={{
                  startAdornment: (
                  <InputAdornment position="start">
                      <SearchIcon />
                  </InputAdornment>
                  ),
                  // Optional: Apply rounded corners directly if theme override isn't
used
                  // sx: { borderRadius: '8px' }
```

```
              }}
              // Optional: Apply rounded corners directly if theme override isn't used
              // sx={{ '& .MuiOutlinedInput-root': { borderRadius: '8px' } }}
            />
            {/* Search Button */}
            <Button
              variant="contained"
              color="primary"
              onClick={handleSearch}
              disabled={loading || !query.trim()}
              sx={{
                  height: '56px',
                  borderRadius: '8px' // Consistent rounded corners
              }}
            >
              {loading ? <CircularProgress size={24} color="inherit" /> : 'Search'}
            </Button>
          </Box>
          {/* Results Display Area */}
          <Box sx={{ mt: 3 }}>
            {/* Loading Indicator */}
            {loading && (
              <Box sx={{ display: 'flex', justifyContent: 'center', my: 3 }}>
                <CircularProgress sx={{ color: 'white' }} /> {/* White spinner for
contrast */}
              </Box>
            )}
            {/* Error Message - Styled for better visibility */}
            {error && (
```

```
              <Alert severity="error" sx={{ mb: 2, borderRadius: '8px' }}>
                {error}
              </Alert>
            )}
            {/* Results Display - Styled for better visibility */}
            {results && !loading && (
              <Box sx={{
                  p: 3,
                  // Kept original style but with rounded corners
                  border: '1px solid rgba(255, 255, 255, 0.3)', // Lighter border
                  borderRadius: '16px', // Rounded corners
                  backgroundColor: 'rgba(255, 255, 255, 0.85)', // Slightly transparent
white
                  backdropFilter: 'blur(3px)', // Slight blur behind results
                  color: '#333' // Darker text color for readability on light bg
              }}>
                <Typography variant="h6" component="h2" gutterBottom>
                  Safety Summary
                </Typography>
                <Typography variant="body1" paragraph>
                  {results.summary}
                </Typography>
                {results.keywords && results.keywords.length > 0 && (
                  <>
                    <Typography variant="h6" component="h3" gutterBottom sx={{ mt: 2 }}>
                      Keywords
                    </Typography>
                    <Box sx={{ display: 'flex', flexWrap: 'wrap', gap: 0.5 }}>
                      {results.keywords.map((keyword, index) => (
```

```
                        <Chip
                          label={keyword}
                          key={index}
                          variant="outlined"
                          size="small"
                          // Optional: Adjust chip color if needed
                          // sx={{ borderColor: 'rgba(0, 0, 0, 0.23)', color: '#333' }}
                        />
                      ))}
                    </Box>
                  </>
                )}
              </Box>
            )}
          </Box>
        </Container>
      </Box>
    </ThemeProvider>
  );
}
export default App;
```

Src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
// You might want a basic CSS reset or global styles
// import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
```

```
  </React.StrictMode>
);
```

src/theme.js

```
import { createTheme } from '@mui/material/styles';

/**
 * Defines the Material UI theme for the Eat Safe application.
 * Includes primary/secondary colors, typography settings, and component overrides.
 */
const theme = createTheme({
  palette: {
    primary: {
      main: '#1976d2', // Blue - Consider changing if it clashes with background
    },
    secondary: {
      main: '#dc004e', // Pink
    },
    background: {
        // Optional: Define a default background if needed, though App.js overrides it
        // default: '#f0f0f0'
    }
  },
  typography: {
    fontFamily: '"Roboto", "Helvetica", "Arial", sans-serif',
    h4: {
      fontWeight: 600,
      marginBottom: '1rem',
    },
    body1: {
```

```
      lineHeight: 1.6,
    },
  },
  components: {
    MuiChip: {
      styleOverrides: {
        root: {
          margin: '4px',
        },
      },
    },
    // Optional: Define default rounded corners for buttons
    MuiButton: {
        styleOverrides: {
            root: {
                borderRadius: '8px', // Example: slightly more rounded buttons by default
            }
        }
    },
    // Optional: Define default rounded corners for text fields
    MuiOutlinedInput: {
        styleOverrides: {
            root: {
                borderRadius: '8px', // Example: slightly more rounded text fields
            }
        }
    }
  },
});
```

```
export default theme;
```

package.json

```
{
  "name": "eat-safe-frontend",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@emotion/react": "^11.11.1",
    "@emotion/styled": "^11.11.0",
    "@mui/icons-material": "^5.14.18",
    "@mui/material": "^5.14.18",
    "axios": "^1.6.2",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
```

```
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```