1. Simon: To create a real-time scraper for public or dark web (Genie GDPR PII auto filtering)

Support Objects: Business strategies, Architectural Stability and Social Network
Adaptive Objects: Feedback Loops, Machine Learning, Threat Intelligence
Core Objects: Designed structures, Human endeavour, Supported Information, Agents, Capabilities

Report

Task

To create a real-time scraper for public or dark web (Genie GDPR PII auto filtering)

PROGRAMS/TOOLS USED

1. Stack overflow
2. Google
3. Google colab

STEP-BY-STEP BREAKDOWN

1. Install Dependencies; We need to install external libraries such as requests, spaCy, beautifulsoup4, and the spaCy English NLP model.
2. User Input: Prompts the user to provide a topic
3. Search Public Sources: DuckDuckGo search API is used to find the top 3 public URLs related to the topic that the user provided
4. Scrape Web Pages: It visits each link and copies the text from the page
5. Detect PII: The content is analyzed through spaCy's NLP pipeline to identify GDPR data.
6. Display Results: For each page, the source, URL, a list of detected PII (with labels), and a snippet of the raw content is presented

```
# ================================
# 🔧 SETUP & INSTALLS
# ================================
!pip install -q requests beautifulsoup4 spacy openai duckduckgo-search
!python -m spacy download en_core_web_sm

import os
import re
import time
import requests
import spacy
from bs4 import BeautifulSoup
from duckduckgo_search import DDGS
from urllib.parse import urlparse

# Load spaCy NLP model for PII detection
nlp = spacy.load("en_core_web_sm")

# ================================
# 🔍 GDPR PII DETECTOR (No Redaction)
# ================================
def detect_gdpr_pii(text):
    doc = nlp(text)
    pii_data = []

    # Named Entities
    for ent in doc.ents:
        if ent.label_ in ["PERSON", "GPE", "ORG", "DATE", "EMAIL",
"PHONE_NUMBER", "LOC", "FAC"]:
            pii_data.append((ent.text, ent.label_))

    # Regex for emails, phones, addresses (rough)
    regex_patterns = {
        "EMAIL": r"[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+",
        "PHONE": r"\+?\d[\d -]{8,}\d",
        "ADDRESS": r"\d{1,5}\s\w+\s\w+",
    }

    for label, pattern in regex_patterns.items():
        matches = re.findall(pattern, text)
```

```
        for match in matches:
            pii_data.append((match, label))

    pii_data = list(set(pii_data))  # Deduplicate
    return pii_data

# ================================
# 🔍 SEARCH + SCRAPE
# ================================
def search_and_scrape(topic, max_results=3):
    print(f"🔎 Searching for: {topic}")
    results = []
    with DDGS() as ddgs:
        search_results = ddgs.text(topic, max_results=max_results)
        for r in search_results:
            url = r['href']
            results.append(url)
    return results

def scrape_page(url):
    try:
        headers = {
            'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)',
            'Accept':
'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.
8',
            'Accept-Language': 'en-US,en;q=0.5',
            'Referer': 'https://www.google.com/'
        }
        res = requests.get(url, headers=headers, timeout=10)

        if res.status_code >= 400:
            print(f"❌ HTTP {res.status_code} for {url}")
            return None  # Skip forbidden pages

        soup = BeautifulSoup(res.content, 'html.parser')
        texts = soup.stripped_strings
        full_text = "\n".join(texts)
        return full_text[:10000]  # Truncate
```

```
    except Exception as e:
        print(f"⚠️ Failed to scrape {url}: {e}")
        return None

# ================================
# 🧠 AGENT - CHAIN OF COMMAND
# ================================
def autonomous_agent(topic):
    urls = search_and_scrape(topic)
    all_outputs = []

    for url in urls:
        print(f"\n🌐 Scraping: {url}")
        raw_text = scrape_page(url)
        if not raw_text or len(raw_text.strip()) < 100:
            print("⚠️ Skipping due to empty or blocked content.")
            continue

        print("🧠 Detecting PII...")
        pii_found = detect_gdpr_pii(raw_text)
        print(f"🔓 {len(pii_found)} PII items detected.")

        domain = urlparse(url).netloc
        output = {
            "source": domain,
            "url": url,
            "pii_count": len(pii_found),
            "pii_items": pii_found,
            "content": raw_text[:2000] + "..."  # Limit display
        }
        all_outputs.append(output)

    return all_outputs

# ================================
# 🚀 RUN
# ================================
topic = input("Enter a public topic to scrape & detect PII (e.g.
'cybersecurity breach 2023'):\n> ")
```

```
results = autonomous_agent(topic)

# ================================
# 📄 DISPLAY RESULTS
# ================================
for i, r in enumerate(results, 1):
    print(f"\n--- RESULT #{i} ---")
    print(f"🔗 Source: {r['source']}")
    print(f"🌍 URL: {r['url']}")
    print(f"🔓 PII Detected: {r['pii_count']}")
    for item, label in r['pii_items']:
        print(f"    - [{label}] {item}")
    print(f"\n📝 Content Snippet:\n{r['content']}")
```

Output

```
🔎 Searching for: data leak

🌐 Scraping: https://haveibeenpwned.com/
🧠 Detecting PII...
🔒 38 PII items detected.

🌐 Scraping: https://www.fortinet.com/resources/cyberglossary/data-leak
🧠 Detecting PII...
🔒 80 PII items detected.

🌐 Scraping: https://www.proofpoint.com/us/threat-reference/data-leak
🧠 Detecting PII...
🔒 67 PII items detected.

--- RESULT #1 ---
🔗 Source: haveibeenpwned.com
🌍 URL: https://haveibeenpwned.com/
🔒 PII Detected: 38
    - [ORG] Telegram
    - [ADDRESS] 099
Combolists Posted
    - [GPE] Exploit
    - [ADDRESS] 627
Troy Hunt
    - [ADDRESS] 1 accounts
763
    - [ADDRESS] 052
Data Enrichment
    - [ADDRESS] 825
paste accounts
    - [ADDRESS] 882
pwned websites
    - [ADDRESS] 011
SpyX accounts
    - [ORG] Verifications.io
    - [ORG] Samsung Germany Customer
    - [PERSON] Generate
    - [ADDRESS] 519
Qraved accounts
    - [ADDRESS] 503
Color Dating
    - [ADDRESS] 3
Subscribe
to
    - [ADDRESS] 698
MySpace accounts
    - [ADDRESS] 643
Spyzie accounts
    - [ADDRESS] 528
Facebook accounts
    - [ADDRESS] 538
Anti Public
    - [PERSON] Moon
    - [ADDRESS] 3 Steps to
    - [ADDRESS] 309
River City
    - [ADDRESS] 078
Boulanger accounts
    - [ADDRESS] 622
```

```
--- RESULT #2 ---
🔗 Source: www.fortinet.com
🌎 URL: https://www.fortinet.com/resources/cyberglossary/data-leak
🔓 PII Detected: 80
    - [ORG] Orange Business Services
    - [ADDRESS] 6 delivers unified
    - [ORG] OT
    - [ORG] Specialized
    - [ORG] Cybercriminals Exploiting New Industry
    - [GPE] Brazil
    - [ORG] NOC Management
Centralized Management
AIOps
AI for
    - [ORG] Cloud AI Workload Protection
Cloud Service Providers
    - [GPE] Korea
    - [ORG] AI-Powered
    - [ORG] Ransomware Advisory Services
Security Advisory Services
    - [ORG] Deutsch
    - [DATE] 1H 2023
    - [PERSON] LTE Wireless WAN

    - [ORG] NAC
    - [ORG] OT
OT Tech Alliance Ecosystem
More
Learn
    - [DATE] 2023
    - [LOC] Mainland China
    - [ORG] Cybersecurity Skills Gap Global Research Report
    - [ORG] SD-Branch
    - [ORG] Financial Services
    - [ORG] Data Leak
    - [ORG] Search
USA
    - [ADDRESS] 6 Common Causes
    - [GPE] France
    - [GPE] Italy
    - [LOC] Latin America
    - [ORG] WLAN
    - [ADDRESS] 2024 Cybersecurity Skills
    - [ORG] IoT
    - [ORG] Wireless
Easily
    - [ORG] State of Operational Technology
    - [ORG] Space-Efficient Security
    - [ORG] Management & Reporting
Secure Access Service Edge
    - [ORG] OT Tech Alliance
OT Security Solutions
Safeguard
    - [ADDRESS] 2023
FortiGuard Labs
    - [ADDRESS] 550 respondents had
    - [ORG] Français
    - [GPE] AI
```

2. Zachary Legal ontology plugin, modality tagging (sarcasm)

Very good paper on this: https://www.ijcai.org/proceedings/2024/0252.pdf
Regulatory: Legitimacy & Framework,
Domain: Socio, Cultural, Technological and Legal,
Strategic: Fake news and strategies
Organizational Culture: Optimisation and Reduction
Organisation: Capability, Innovation, and Leadership
Organisational Strategy: Social Media and Facts
Individual: Demographic, Personality, Self-Efficacy, Belief and Trust
Information Source: Information Ecosystem, Risk, and Propagation

Report

Task

To create a Legal ontology plugin with modality tagging

PROGRAMS/TOOLS USED

1. Stack overflow
2. Google
3. Google colab

STEP-BY-STEP BREAKDOWN

1. Download the required packages. For this task several packages were needed to be downloaded. nltk spacy textblob scikit-learn networkx matplotlib seaborn pandas were needed to be downloaded.
2. Creating a class for LegalOntologyPlugin allows for legal ontology caretgories and modality types to be imported for sorting words to the appropriate modality marker
3. Common words for each modality marker are listed to identify which words in the added text associate with each modality.
4. For the sarcasm detector, some sample sentences were added to distinguish between sarcastic and non-sarcastic comments.
5. In order to distinguish the categories for the text that was input 'analyze_legal_text' to differentiate between modality markers, sarcasm and named entities.
6. To create the graph, networkx allows for the data to be represented visually, separating modality types, and categories.
7. To create a visual representation for modality distribution, matploitlib was used to show the amount of times each modality type was identified
8. Generating the report required the use of generate_report, which will also display the graphs

```
# Complete Legal Ontology Plugin with Enhanced Modality Tagging
# For Google Colab

# Install required packages
!pip install nltk spacy textblob scikit-learn networkx matplotlib seaborn pandas

# Download spaCy language model
!python -m spacy download en_core_web_sm

# Create the plugin code
import nltk
import spacy
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from nltk.sentiment import SentimentIntensityAnalyzer
from textblob import TextBlob
import re
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report
import networkx as nx

class LegalOntologyPlugin:
    """
    A plugin for analyzing legal texts with ontology mapping and comprehensive modality
detection
    including sarcasm detection capabilities.
    """

    def __init__(self):
        """Initialize the plugin by installing and loading required resources"""
        # Download required NLTK data
        try:
            nltk.data.find('tokenizers/punkt')
```

```
        except LookupError:
            nltk.download('punkt')

        try:
            nltk.data.find('corpora/stopwords')
        except LookupError:
            nltk.download('stopwords')

        try:
            nltk.data.find('sentiment/vader_lexicon.zip')
        except LookupError:
            nltk.download('vader_lexicon')

        # Load spaCy model
        try:
            self.nlp = spacy.load('en_core_web_sm')
        except:
            # If model not installed, download it
            import os
            os.system('python -m spacy download en_core_web_sm')
            self.nlp = spacy.load('en_core_web_sm')

        # Initialize sentiment analyzer
        self.sia = SentimentIntensityAnalyzer()

        # Legal term ontology (simplified)
        self.legal_ontology = {
            'contract': ['agreement', 'covenant', 'obligation', 'consideration', 'term'],
            'tort': ['negligence', 'liability', 'damages', 'injury', 'duty of care'],
            'property': ['real estate', 'ownership', 'title', 'deed', 'easement'],
            'criminal': ['offense', 'prosecution', 'defendant', 'guilty', 'sentence'],
            'constitutional': ['rights', 'amendment', 'liberty', 'freedom', 'due process'],
            'procedural': ['motion', 'pleading', 'discovery', 'evidence', 'testimony']
        }

        # Basic modality markers
        self.modality_markers = {
            'certainty': ['definitely', 'certainly', 'always', 'must', 'will', 'shall'],
            'probability': ['likely', 'probably', 'may', 'might', 'could', 'possibly'],
```

```
        'permission': ['may', 'can', 'allowed', 'permitted', 'authorized'],
        'obligation': ['must', 'should', 'shall', 'required', 'obligated', 'necessary'],
        'intention': ['will', 'going to', 'intend', 'plan', 'aim'],
        'ability': ['can', 'could', 'able to', 'capability', 'competence']
    }

    # Enhanced modalities
    # Epistemic modalities (related to knowledge and belief)
    self.modality_markers['epistemic'] = [
        'know', 'believe', 'think', 'suppose', 'assume', 'presume',
        'appears that', 'seems that', 'is known', 'is believed',
        'allegedly', 'reportedly', 'ostensibly', 'purportedly',
        'evidently', 'apparently'
    ]

    # Deontic modalities (related to duty and obligation beyond the basics)
    self.modality_markers['deontic_prohibition'] = [
        'prohibited', 'forbidden', 'must not', 'shall not', 'may not',
        'disallowed', 'impermissible', 'banned', 'barred', 'illegal',
        'unlawful', 'impermissible', 'restricted'
    ]

    # Alethic modalities (related to logical necessity and possibility)
    self.modality_markers['alethic'] = [
        'necessarily', 'necessarily true', 'logically entails',
        'impossible', 'logically impossible', 'by definition',
        'axiomatically', 'tautologically', 'inherently'
    ]

    # Boulomaic modalities (related to desire and preference)
    self.modality_markers['boulomaic'] = [
        'hope', 'wish', 'want', 'prefer', 'desire',
        'ideally', 'preferably', 'hopefully',
        'it is desired that', 'it is preferred that'
    ]

    # Temporal modalities (related to time)
    self.modality_markers['temporal'] = [
        'always', 'never', 'sometimes', 'occasionally',
```

```
        'henceforth', 'thereafter', 'previously', 'subsequently',
        'heretofore', 'hereinafter', 'forthwith', 'instanter'
    ]

    # Evidential modalities (related to evidence and proof)
    self.modality_markers['evidential'] = [
        'proves', 'demonstrates', 'shows', 'indicates', 'suggests',
        'evidences', 'establishes', 'verifies', 'confirms',
        'prima facie', 'beyond reasonable doubt', 'on the balance of probabilities',
        'circumstantial evidence', 'direct evidence', 'hearsay'
    ]

    # Conditional modalities (if-then relationships)
    self.modality_markers['conditional'] = [
        'if', 'unless', 'provided that', 'on condition that',
        'subject to', 'contingent upon', 'in the event that',
        'assuming that', 'in case', 'should', 'were'
    ]

    # Dispositional modalities (related to capabilities and tendencies)
    self.modality_markers['dispositional'] = [
        'prone to', 'disposed to', 'tends to', 'inclined to',
        'liable to', 'susceptible to', 'predisposed to'
    ]

    # Volitive modalities (related to willfulness)
    self.modality_markers['volitive'] = [
        'willfully', 'intentionally', 'deliberately', 'knowingly',
        'purposefully', 'consciously', 'voluntarily'
    ]

    # Intensional modalities (affecting interpretation)
    self.modality_markers['intensional'] = [
        'construed as', 'interpreted as', 'understood as', 'deemed to be',
        'considered to be', 'regarded as', 'treated as'
    ]

    # Train a basic sarcasm detector
    self._train_sarcasm_detector()
```

```
    print("Legal Ontology Plugin initialized successfully with enhanced modalities!")

def _train_sarcasm_detector(self):
    """Train a simple sarcasm detector"""
    # Create a simple dataset for demonstration
    sarcastic_phrases = [
        "Oh, that's exactly what we need, more legal jargon.",
        "Wow, another brilliant legal interpretation.",
        "Sure, that contract is totally clear and straightforward.",
        "The judge was definitely impressed with that argument.",
        "Yeah, the law is so simple to understand.",
        "Absolutely, legal writing is known for its clarity.",
        "Oh great, another 50-page document to review.",
        "This case is clearly going to be resolved quickly.",
        "The brevity of legal documents is their best feature.",
        "I'm thrilled to read another terms of service agreement."
    ]

    non_sarcastic_phrases = [
        "The contract specifies payment terms in section 3.",
        "The court ruled in favor of the plaintiff.",
        "Legal precedent suggests this interpretation is correct.",
        "The statute clearly defines the requirements.",
        "Both parties have signed the agreement.",
        "The evidence supports the prosecution's case.",
        "This document requires notarization.",
        "The law prohibits this type of conduct.",
        "We need to file the motion by Friday.",
        "The terms are outlined in the attached document."
    ]

    # Combine and label
    texts = sarcastic_phrases + non_sarcastic_phrases
    labels = [1] * len(sarcastic_phrases) + [0] * len(non_sarcastic_phrases)

    # Create features
    self.vectorizer = CountVectorizer(stop_words='english')
    X = self.vectorizer.fit_transform(texts)
```

```
    # Train the model
    self.sarcasm_model = MultinomialNB()
    self.sarcasm_model.fit(X, labels)

def detect_sarcasm(self, text):
    """Detect if text contains sarcasm"""
    # Transform the text
    text_features = self.vectorizer.transform([text])

    # Predict
    prediction = self.sarcasm_model.predict(text_features)[0]
    probability = self.sarcasm_model.predict_proba(text_features)[0][1]

    return {
        'is_sarcastic': bool(prediction),
        'sarcasm_probability': probability,
        'interpretation': "Likely sarcastic" if probability > 0.5 else "Likely literal"
    }

def analyze_legal_text(self, text):
    """Analyze legal text for ontology mapping and modality"""
    doc = self.nlp(text)

    # Analyze sentiment
    sentiment = self.sia.polarity_scores(text)

    # Find legal concepts
    legal_concepts = {}
    for category, terms in self.legal_ontology.items():
        legal_concepts[category] = []
        for term in terms:
            if term in text.lower():
                legal_concepts[category].append(term)

    # Detect modalities
    modalities = {}
    for modality, markers in self.modality_markers.items():
        modalities[modality] = []
```

```
            for marker in markers:
                if marker in text.lower():
                    modalities[modality].append(marker)

        # Detect sarcasm
        sarcasm = self.detect_sarcasm(text)

        # Extract named entities
        entities = [(ent.text, ent.label_) for ent in doc.ents]

        # Detect modality relationships
        modality_relationships = self.detect_modality_relationships(text)

        return {
            'sentiment': sentiment,
            'legal_concepts': legal_concepts,
            'modalities': modalities,
            'sarcasm': sarcasm,
            'entities': entities,
            'sentence_count': len(list(doc.sents)),
            'word_count': len([token for token in doc if not token.is_punct]),
            'modality_relationships': modality_relationships
        }

    def detect_modality_relationships(self, text):
        """
        Detect relationships between different modalities in the text

        Parameters:
        text -- The legal text to analyze

        Returns:
        A dictionary of modality relationships and their instances
        """
        # Define common modality combinations in legal texts
        modality_relationships = {
            'conditional_obligation': {
                'pattern':
r'if\s+.{1,50}?\s+must|if\s+.{1,50}?\s+shall|provided\s+that\s+.{1,50}?\s+required',
```

```
                'instances': []
            },
            'permissive_exception': {
                'pattern':
r'except\s+.{1,50}?\s+may|unless\s+.{1,50}?\s+can|notwithstanding\s+.{1,50}?\s+permitted',
                'instances': []
            },
            'epistemic_probability': {
                'pattern':
r'know\s+.{1,50}?\s+likely|believe\s+.{1,50}?\s+probably|think\s+.{1,50}?\s+may',
                'instances': []
            },
            'temporal_obligation': {
                'pattern':
r'always\s+.{1,50}?\s+must|never\s+.{1,50}?\s+shall|thereafter\s+.{1,50}?\s+required',
                'instances': []
            },
            'evidential_certainty': {
                'pattern':
r'proves\s+.{1,50}?\s+certainly|demonstrates\s+.{1,50}?\s+definitely|evidence\s+.{1,50}?\s+conc
lusively',
                'instances': []
            },
            'prohibitive_consequence': {
                'pattern': r'prohibited\s+.{1,50}?\s+if|forbidden\s+.{1,50}?\s+when|shall
not\s+.{1,50}?\s+unless',
                'instances': []
            },
            'interpretive_obligation': {
                'pattern':
r'construed\s+.{1,50}?\s+must|interpreted\s+.{1,50}?\s+shall|deemed\s+.{1,50}?\s+required',
                'instances': []
            }
        }

        # Find instances of each relationship pattern
        for rel_name, rel_data in modality_relationships.items():
            matches = re.finditer(rel_data['pattern'], text.lower())
            for match in matches:
```

```
                rel_data['instances'].append(match.group(0))

        return modality_relationships

    def visualize_legal_ontology(self, text):
        """Visualize the legal ontology network based on text analysis"""
        # Analyze text
        analysis = self.analyze_legal_text(text)

        # Create graph
        G = nx.Graph()

        # Add nodes for found legal concepts
        for category, terms in analysis['legal_concepts'].items():
            if terms:  # If terms found for this category
                G.add_node(category, size=100, color='blue')
                for term in terms:
                    G.add_node(term, size=50, color='green')
                    G.add_edge(category, term)

        # Add modality connections if they exist
        for modality, markers in analysis['modalities'].items():
            if markers:
                G.add_node(modality, size=75, color='red')
                for category in G.nodes():
                    if category in analysis['legal_concepts']:
                        G.add_edge(modality, category, style='dashed')

        # Add modality relationship nodes
        for rel_name, rel_data in analysis['modality_relationships'].items():
            if rel_data['instances']:
                G.add_node(rel_name, size=85, color='purple')
                # Connect to relevant modalities
                rel_parts = rel_name.split('_')
                for part in rel_parts:
                    for modality in G.nodes():
                        if part in modality and modality in analysis['modalities']:
                            G.add_edge(rel_name, modality, style='dotted')
```

```
        # Visualize
        plt.figure(figsize=(12, 10))
        pos = nx.spring_layout(G, seed=42)  # For consistent layout
        node_colors = [G.nodes[node].get('color', 'gray') for node in G.nodes()]
        node_sizes = [G.nodes[node].get('size', 30) for node in G.nodes()]

        nx.draw_networkx(
            G, pos,
            node_color=node_colors,
            node_size=node_sizes,
            font_size=10,
            edge_color='gray',
            with_labels=True
        )

        plt.title("Legal Ontology Network with Modalities")
        plt.axis('off')
        return plt

    def visualize_modality_distribution(self, text):
        """Visualize the distribution of modalities in the text"""
        analysis = self.analyze_legal_text(text)

        # Count modalities
        modality_counts = {m: len(instances) for m, instances in analysis['modalities'].items() if
instances}

        if not modality_counts:
            print("No modalities detected in the text.")
            return None

        # Create visualization
        plt.figure(figsize=(10, 6))
        bars = plt.bar(modality_counts.keys(), modality_counts.values(), color='skyblue')
        plt.title('Distribution of Modalities in Text')
        plt.xlabel('Modality Type')
        plt.ylabel('Frequency')
        plt.xticks(rotation=45, ha='right')
        plt.tight_layout()
```

```
    # Add count labels
    for bar in bars:
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width()/2., height + 0.1,
                f'{height}', ha='center', va='bottom')

    return plt

def generate_report(self, text):
    """Generate a comprehensive report on the legal text"""
    analysis = self.analyze_legal_text(text)

    print("===== LEGAL TEXT ANALYSIS REPORT =====")
    print(f"Word count: {analysis['word_count']}")
    print(f"Sentence count: {analysis['sentence_count']}")
    print("\n--- SENTIMENT ANALYSIS ---")
    print(f"Positive: {analysis['sentiment']['pos']:.2f}")
    print(f"Neutral: {analysis['sentiment']['neu']:.2f}")
    print(f"Negative: {analysis['sentiment']['neg']:.2f}")
    print(f"Compound: {analysis['sentiment']['compound']:.2f}")

    print("\n--- LEGAL CONCEPTS DETECTED ---")
    for category, terms in analysis['legal_concepts'].items():
        if terms:
            print(f"{category.capitalize()}: {', '.join(terms)}")

    print("\n--- MODALITY ANALYSIS ---")
    for modality, markers in analysis['modalities'].items():
        if markers:
            print(f"{modality.capitalize()}: {', '.join(markers)}")

    print("\n--- MODALITY RELATIONSHIPS ---")
    for rel_name, rel_data in analysis['modality_relationships'].items():
        if rel_data['instances']:
            print(f"{rel_name.replace('_', ' ').title()}:")
            for instance in rel_data['instances']:
                print(f"  - \"{instance}\"")
```

```
        print("\n--- SARCASM DETECTION ---")
        print(f"Interpretation: {analysis['sarcasm']['interpretation']}")
        print(f"Sarcasm probability: {analysis['sarcasm']['sarcasm_probability']:.2f}")

        print("\n--- NAMED ENTITIES ---")
        for entity, label in analysis['entities']:
            print(f"{entity} ({label})")

        # Generate visualizations
        print("\n--- VISUALIZATIONS ---")
        print("Generating Legal Ontology Network...")
        ont_plt = self.visualize_legal_ontology(text)
        ont_plt.show()

        print("Generating Modality Distribution Chart...")
        mod_plt = self.visualize_modality_distribution(text)
        if mod_plt:
            mod_plt.show()

        return analysis

# Create a demo function to run in Google Colab
def run_legal_ontology_demo():
    # Initialize the plugin
    plugin = LegalOntologyPlugin()

    # Test with a complex legal text that uses various modalities
    complex_legal_text = """
    The Court finds that the plaintiff has prima facie established a claim of negligence.
    If the defendant knew or should have known of the dangerous condition, they must take
reasonable steps to remedy it.
    The contract shall be construed as binding unless otherwise explicitly provided.
    Heretofore, the parties were allegedly operating under an implied agreement.
    The witness testimony suggests that the defendant willfully disregarded the warning signs.
    The statute evidently prohibits such conduct, and it is necessarily true that violations thereof
are actionable.
    It is desired that the parties attempt mediation before proceeding to trial.
    The tenant is hereby prohibited from subletting the premises without prior written consent.
    """
```

```
# Sample sarcastic legal text
sarcastic_text = """
Oh sure, this 200-page contract is absolutely crystal clear.
I'm thrilled to spend my weekend reviewing these fascinating legal citations.
The judge was definitely convinced by that brilliant argument.
The language of this statute is just a model of clarity, isn't it?
I'm absolutely certain that all parties must completely understand these terms.
"""

# Analyze regular text
print("\n=== ANALYZING COMPLEX LEGAL TEXT ===")
analysis = plugin.analyze_legal_text(complex_legal_text)
print("Modalities detected:")
for modality_type, instances in analysis['modalities'].items():
    if instances:
        print(f"- {modality_type}: {', '.join(instances)}")

# Generate full report for complex text
print("\n=== GENERATING FULL REPORT FOR COMPLEX TEXT ===")
plugin.generate_report(complex_legal_text)

# Analyze sarcastic text
print("\n=== ANALYZING SARCASTIC LEGAL TEXT ===")
sarcasm_analysis = plugin.analyze_legal_text(sarcastic_text)
print(f"Sarcasm analysis: {sarcasm_analysis['sarcasm']}")

# Generate full report for sarcastic text
print("\n=== GENERATING FULL REPORT FOR SARCASTIC TEXT ===")
plugin.generate_report(sarcastic_text)

# Set up interactive interface
try:
    import ipywidgets as widgets
    from IPython.display import display

    input_text = widgets.Textarea(
        value='',
        placeholder='Enter legal text to analyze...',
```

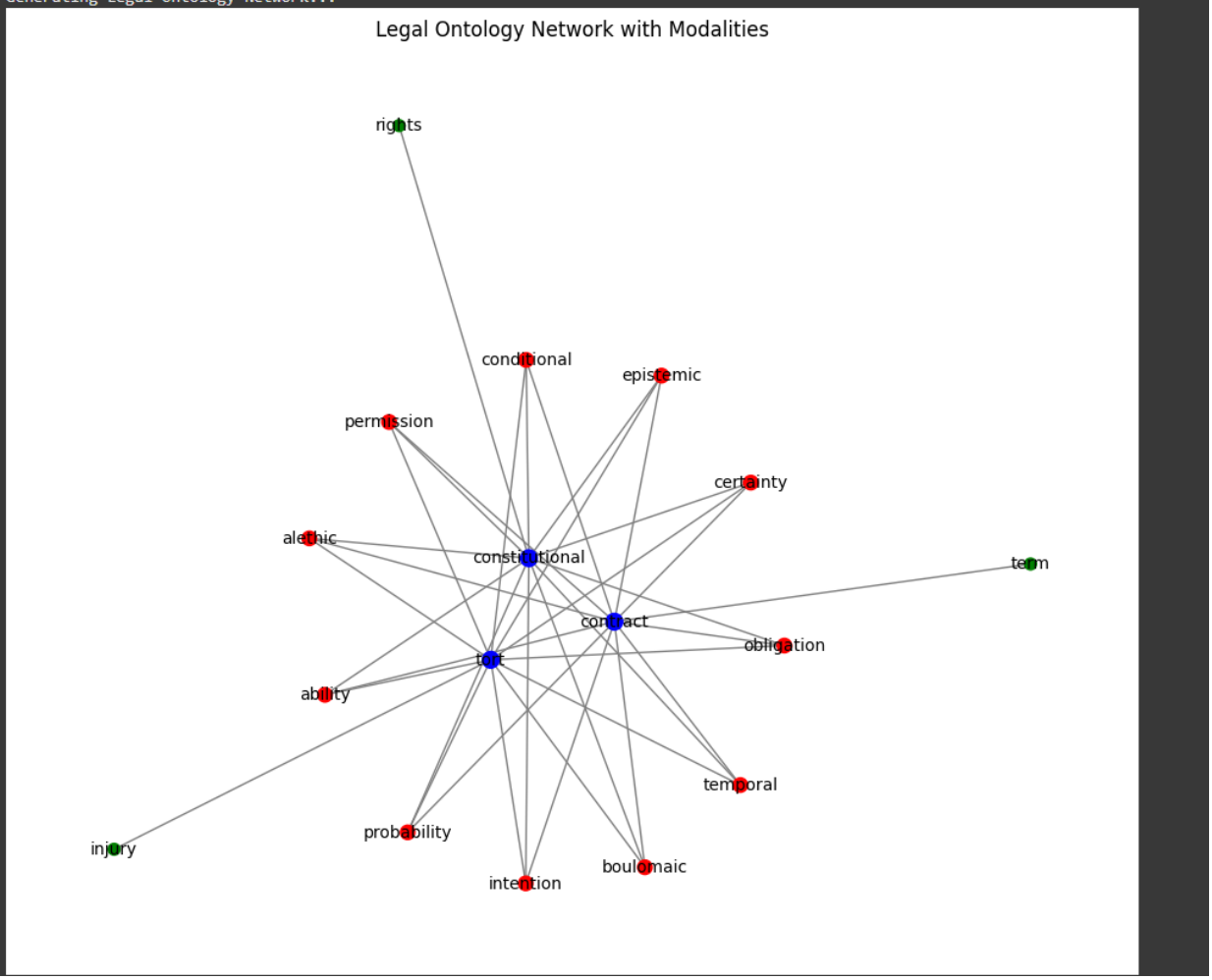
```
            description='Text:',
            disabled=False,
            layout=widgets.Layout(width='100%', height='150px')
        )

        analyze_button = widgets.Button(
            description='Analyze Text',
            disabled=False,
            button_style='',
            tooltip='Click to analyze the legal text',
            icon='check'
        )

        output = widgets.Output()

        def on_button_clicked(b):
            with output:
                output.clear_output()
                if input_text.value:
                    print("Analyzing text...")
                    plugin.generate_report(input_text.value)
                else:
                    print("Please enter some text to analyze.")

        analyze_button.on_click(on_button_clicked)

        print("\n=== INTERACTIVE LEGAL TEXT ANALYZER ===")
        print("Enter legal text and click 'Analyze Text' to generate a report.")
        display(input_text)
        display(analyze_button)
        display(output)

    except ImportError:
        print("\nNOTE: Interactive widgets not available. Installing ipywidgets...")
        !pip install ipywidgets
        print("Please restart the runtime and run again to use the interactive interface.")

# Run the demo if this script is executed directly
if __name__ == "__main__":
```

```
run_legal_ontology_demo()
```

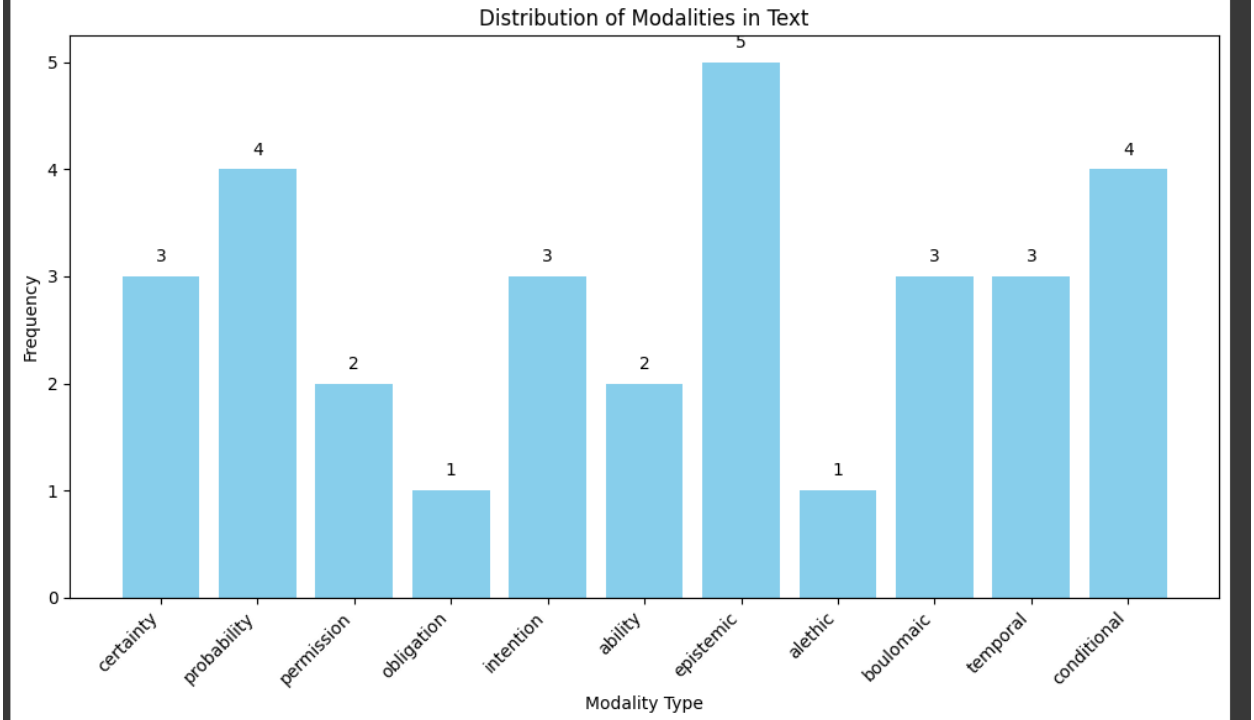
Output

```
Analyzing text...
===== LEGAL TEXT ANALYSIS REPORT =====
Word count: 6500
Sentence count: 574

--- SENTIMENT ANALYSIS ---
Positive: 0.08
Neutral: 0.87
Negative: 0.04
Compound: 1.00

--- LEGAL CONCEPTS DETECTED ---
Contract: term
Tort: injury
Constitutional: rights

--- MODALITY ANALYSIS ---
Certainty: definitely, always, will
Probability: probably, may, might, could
Permission: may, can
Obligation: should
Intention: will, going to, plan
Ability: can, could
Epistemic: know, believe, think, suppose, evidently
Alethic: necessarily
Boulomaic: wish, want, desire
Temporal: always, never, sometimes
Conditional: if, unless, should, were

--- MODALITY RELATIONSHIPS ---

--- SARCASM DETECTION ---
Interpretation: Likely sarcastic
Sarcasm probability: 1.00

--- NAMED ENTITIES ---
Detective Lange (PERSON)
Parker Center (GPE)
June 13th, 1994 (DATE)
13:35 hours (TIME)
O.J. Simpson (PERSON)
Orenthal James Simpson (PERSON)
Simpson (PERSON)
James Simpson (PERSON)
Simpson (PERSON)
Simpson (PERSON)
July 9th, 1947 (DATE)
Simpson (PERSON)
Simpson (PERSON)
Simpson (PERSON)
```

```
--- VISUALIZATIONS ---
Generating Legal Ontology Network...
```



Generating Modality Distribution Chart...



3. Imran: Auto detect legal loopholes, or compliance conflicts, heat map of high conflict nodes

Misinformation, things shared without intent to harm
Disinformation, things created to deceive

Report

Task

To create a tool that can auto detect legal loopholes, or compliance conflicts, heat map of high conflict nodes

PROGRAMS/TOOLS USED

1. Stack overflow
2. Google
3. Google colab

STEP-BY-STEP BREAKDOWN

1. Download the required files and libraries necessary for the tool to work, libraries that were require were: transformers, wordcloud, networkx, nltk, pandas, seaborn, plotly, and spaCy.
2. In order to analyze legal compliance, LegalComplianceAnalyzer is created as a class to provide a list of legal terms, and to implement a transformer.
3. To detect conflicts, detect_potential_conflicts uses TF-IDF and Cosine to determine keywords that may indicate a potential conflict
4. identify_loopholes, allows for patterns to identify specific phrases that may be a part of a loophole.
5. In order to identify compliance conflicts, analyze_compliance_conflicts is used to cross match keywords to conflict scores based on loop holes.
6. To visualise the data, a heatmap and network graph were added to highlight conflict scores and conflict indicators
7. generate_conflict_report, allows for a summary of the data to be placed in a table
8. highlight_document_conflicts, creates a HTML document with the conflicts highlighted by severity

```
# Legal Compliance Conflict Detection Tool
# A tool to analyze legal documents, detect potential compliance
conflicts,
# and visualize high-conflict areas through heat maps

# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import networkx as nx
import nltk
import re
import spacy
import os
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from transformers import AutoTokenizer, AutoModelForSequenceClassification
import torch
from wordcloud import WordCloud
from tqdm.notebook import tqdm
import plotly.graph_objects as go
import plotly.express as px

# Install required packages
!pip install -q spacy transformers wordcloud networkx nltk pandas seaborn
plotly
!python -m spacy download en_core_web_sm

# Download NLTK data
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')

class LegalComplianceAnalyzer:
    def __init__(self):
        """Initialize the Legal Compliance Analyzer with necessary NLP
models and data structures."""
        # Load NLP model
        print("Loading NLP models...")
```

```
        self.nlp = spacy.load("en_core_web_sm")

        # Define common legal terms that might indicate conflicts or
loopholes
        self.conflict_terms = [
            "notwithstanding", "subject to", "except as", "unless",
"however",
            "provided that", "but", "despite", "although", "exempt",
"waiver",
            "override", "supersede", "contradict", "conflict",
"inconsistent",
            "ambiguous", "vague", "unclear", "discretion", "reasonably",
"interpretation"
        ]

        # Define common legal sections for grouping
        self.legal_sections = [
            "definition", "scope", "term", "termination", "payment",
"confidentiality",
            "intellectual property", "warranty", "liability",
"indemnification",
            "governing law", "dispute", "amendment", "force majeure",
"notice",
            "assignment", "severability", "waiver", "entire agreement"
        ]

        # Prepare the conflict detection model
        self.tokenizer = AutoTokenizer.from_pretrained("distilbert-base-
uncased")
        self.model =
AutoModelForSequenceClassification.from_pretrained("distilbert-base-
uncased", num_labels=2)

        print("Analyzer initialized successfully.")

    def load_document(self, file_path=None, text=None):
        """Load document from file or text string."""
        if file_path:
            with open(file_path, 'r', encoding='utf-8') as file:
                self.document_text = file.read()
```

```
        elif text:
            self.document_text = text
        else:
            raise ValueError("Either file_path or text must be provided")

        # Process the document
        self.doc = self.nlp(self.document_text)

        # Split into sections and paragraphs
        self.sections = self._split_into_sections()
        self.paragraphs = self._split_into_paragraphs()

        print(f"Document loaded: {len(self.paragraphs)} paragraphs
identified.")
        return self.document_text

    def _split_into_sections(self):
        """Split document into legal sections based on headers."""
        # Simple section split based on capitalized headers or numbered
sections
        section_pattern = r'((?:\d+\.)+\s+[A-Z][^.]*\.|[A-Z][A-Z\s]+:)'
        sections = re.split(section_pattern, self.document_text)

        # Clean up and pair headers with content
        result = []
        for i in range(1, len(sections), 2):
            if i < len(sections)-1:
                header = sections[i].strip()
                content = sections[i+1].strip()
                result.append((header, content))

        # If no sections found with the pattern, use paragraph breaks
        if not result:
            paragraphs = self.document_text.split('\n\n')
            for p in paragraphs:
                if p.strip():
                    result.append(("Paragraph", p.strip()))

        return result
```

```
    def _split_into_paragraphs(self):
        """Split document into paragraphs for analysis."""
        paragraphs = []
        for section_header, section_content in self.sections:
            for para in section_content.split('\n\n'):
                if para.strip():
                    paragraphs.append({
                        'section': section_header,
                        'content': para.strip(),
                        'entities': [],
                        'conflicts': [],
                        'conflict_score': 0
                    })

        # Process each paragraph with spaCy
        for para in paragraphs:
            doc = self.nlp(para['content'])
            para['entities'] = [(ent.text, ent.label_) for ent in
doc.ents]

        return paragraphs

    def detect_potential_conflicts(self):
        """Detect potential conflicts in the document."""
        print("Detecting potential conflicts...")

        # TF-IDF for similarity analysis
        vectorizer = TfidfVectorizer(stop_words='english')
        try:
            tfidf_matrix = vectorizer.fit_transform([p['content'] for p in
self.paragraphs])
            similarity_matrix = cosine_similarity(tfidf_matrix)
        except ValueError:  # In case there's not enough content
            similarity_matrix = np.zeros((len(self.paragraphs),
len(self.paragraphs)))

        # Network to track relationships
        G = nx.Graph()
        for i, para in enumerate(self.paragraphs):
```

```
            G.add_node(i, content=para['content'][:50] + "...",
section=para['section'])

        # Conflict detection for each paragraph
        for i, para in enumerate(tqdm(self.paragraphs)):
            para_content = para['content'].lower()

            # Check for conflict terms
            conflict_indicators = [term for term in self.conflict_terms if
term in para_content]
            para_conflict_score = len(conflict_indicators) * 0.5

            # Check for related paragraphs that may conflict
            related_paragraphs = []
            for j, similarity in enumerate(similarity_matrix[i]):
                if i != j and similarity > 0.3:  # Paragraphs with some
similarity
                    related_paragraphs.append((j, similarity))
                    G.add_edge(i, j, weight=similarity)

            # For each related paragraph, check for potential conflicts
            for j, similarity in related_paragraphs:
                related_para = self.paragraphs[j]

                # Check if they're about the same topic but have different
requirements
                conflict_terms_in_related = [term for term in
self.conflict_terms if term in related_para['content'].lower()]

                # If both paragraphs have conflict terms, strengthen the
score
                if conflict_indicators and conflict_terms_in_related:
                    para_conflict_score += 0.5

                # Subject analysis (do they refer to the same entities?)
                para_entities = set([e[0].lower() for e in
para['entities']])
                related_entities = set([e[0].lower() for e in
related_para['entities']])
```

```
                common_entities =
para_entities.intersection(related_entities)

                if common_entities:
                    para_conflict_score += 0.3 * len(common_entities)

                    # Add this as a potential conflict
                    conflict_info = {
                        'related_para_idx': j,
                        'similarity': similarity,
                        'common_entities': list(common_entities),
                        'conflict_terms': conflict_indicators +
conflict_terms_in_related
                    }
                    para['conflicts'].append(conflict_info)

            # Record the final conflict score
            para['conflict_score'] = min(para_conflict_score, 10)  # Cap
at 10

            # Add conflicting terms found to nodes
            G.nodes[i]['conflict_score'] = para['conflict_score']

        self.conflict_graph = G
        print(f"Conflict detection complete. Found {sum(1 for p in
self.paragraphs if p['conflict_score'] > 2)} paragraphs with significant
conflict potential.")
        return self.paragraphs

    def identify_loopholes(self):
        """Identify potential legal loopholes in the document."""
        print("Identifying potential loopholes...")

        # Look for patterns that may indicate loopholes
        loophole_patterns = [
            r"except\s+(?:in|as|when|under|for)",
            r"unless\s+otherwise",
            r"at\s+(\w+)(\s+)discretion",
            r"reasonable\s+efforts",
            r"commercially\s+reasonable",
```

```
            r"may\s+(?:be|have)",
            r"not\s+obligated\s+to",
            r"no\s+obligation\s+to",
            r"sole\s+discretion"
        ]

        loopholes = []

        for i, para in enumerate(self.paragraphs):
            para_loopholes = []
            for pattern in loophole_patterns:
                matches = re.finditer(pattern, para['content'].lower())
                for match in matches:
                    # Get some context around the match
                    start = max(0, match.start() - 50)
                    end = min(len(para['content']), match.end() + 50)
                    context = para['content'][start:end]

                    para_loopholes.append({
                        'pattern': pattern,
                        'match': match.group(0),
                        'context': context
                    })

            if para_loopholes:
                loopholes.append({
                    'paragraph_idx': i,
                    'section': para['section'],
                    'loopholes': para_loopholes,
                    'count': len(para_loopholes)
                })
                # Update the paragraph with loophole info
                self.paragraphs[i]['loopholes'] = para_loopholes
                self.paragraphs[i]['loophole_count'] = len(para_loopholes)
                # Add to conflict score
                self.paragraphs[i]['conflict_score'] += 
min(len(para_loopholes), 5)

        self.loopholes = loopholes
```

```
        print(f"Loophole detection complete. Found {len(loopholes)}
paragraphs with potential loopholes.")
        return loopholes

    def analyze_compliance_conflicts(self, regulations=None):
        """Analyze potential compliance conflicts with specified
regulations."""
        if not regulations:
            print("No specific regulations provided for compliance
analysis.")
            return []

        print("Analyzing compliance conflicts...")

        compliance_conflicts = []
        for i, para in enumerate(self.paragraphs):
            para_conflicts = []

            for reg in regulations:
                # Simple keyword matching for now
                if any(keyword in para['content'].lower() for keyword in
reg['keywords']):
                    # This paragraph mentions topics related to the
regulation
                    conflict_likelihood = 0.5  # Base likelihood

                    # If paragraph has loopholes, increase likelihood
                    if 'loopholes' in para and para['loopholes']:
                        conflict_likelihood += 0.3

                    # If paragraph has high conflict score, increase
likelihood
                    if para['conflict_score'] > 5:
                        conflict_likelihood += 0.2

                    para_conflicts.append({
                        'regulation': reg['name'],
                        'keywords_found': [k for k in reg['keywords'] if k
in para['content'].lower()],
                        'conflict_likelihood': conflict_likelihood
```

```
                    })

            if para_conflicts:
                compliance_conflicts.append({
                    'paragraph_idx': i,
                    'section': para['section'],
                    'conflicts': para_conflicts
                })
                # Update paragraph with compliance info
                self.paragraphs[i]['compliance_conflicts'] =
para_conflicts

        self.compliance_conflicts = compliance_conflicts
        print(f"Compliance analysis complete. Found
{len(compliance_conflicts)} paragraphs with potential compliance issues.")
        return compliance_conflicts

    def generate_conflict_heatmap(self):
        """Generate a heatmap of conflict areas in the document."""
        print("Generating conflict heatmap...")

        # Extract section and conflict scores
        sections = [p['section'] for p in self.paragraphs]
        conflict_scores = [p['conflict_score'] for p in self.paragraphs]

        # Get unique sections to group by
        unique_sections = list(set(sections))
        section_scores = {}

        for section in unique_sections:
            indices = [i for i, s in enumerate(sections) if s == section]
            avg_score = sum(conflict_scores[i] for i in indices) /
len(indices)
            section_scores[section] = avg_score

        # Prepare data for the heatmap
        heatmap_data = pd.DataFrame({
            'Section': list(section_scores.keys()),
            'Conflict Score': list(section_scores.values())
        })
```

```
        # Sort by conflict score
        heatmap_data = heatmap_data.sort_values('Conflict Score',
ascending=False)

        # Create heatmap using Plotly for better interactivity in Colab
        fig = px.bar(
            heatmap_data,
            x='Section',
            y='Conflict Score',
            color='Conflict Score',
            color_continuous_scale='Reds',
            title='Document Section Conflict Heatmap'
        )

        fig.update_layout(
            xaxis_title='Document Section',
            yaxis_title='Conflict Score',
            coloraxis_colorbar=dict(title='Conflict Severity')
        )

        # Also create paragraph-level heatmap
        para_data = pd.DataFrame({
            'Paragraph': range(len(self.paragraphs)),
            'Section': sections,
            'Conflict Score': conflict_scores
        })

        fig2 = px.scatter(
            para_data,
            x='Paragraph',
            y='Section',
            color='Conflict Score',
            size='Conflict Score',
            color_continuous_scale='Reds',
            title='Paragraph-level Conflict Heatmap'
        )

        fig2.update_layout(
            xaxis_title='Paragraph Index',
```

```
            yaxis_title='Document Section',
            coloraxis_colorbar=dict(title='Conflict Severity')
        )

        return fig, fig2

    def generate_network_graph(self):
        """Generate a network graph of conflicts between paragraphs."""
        print("Generating conflict network graph...")

        G = self.conflict_graph

        # Get positions for nodes
        pos = nx.spring_layout(G, seed=42)

        # Get node colors based on conflict score
        node_colors = [G.nodes[n].get('conflict_score', 0) for n in
G.nodes()]

        # Get node sizes based on degree
        node_sizes = [300 * (1 + G.degree(n)) for n in G.nodes()]

        # Get edge weights
        edge_weights = [G[u][v].get('weight', 0.1) * 3 for u, v in
G.edges()]

        # Create Plotly network graph
        edge_x = []
        edge_y = []
        for edge in G.edges():
            x0, y0 = pos[edge[0]]
            x1, y1 = pos[edge[1]]
            edge_x.extend([x0, x1, None])
            edge_y.extend([y0, y1, None])

        edge_trace = go.Scatter(
            x=edge_x, y=edge_y,
            line=dict(width=0.5, color='#888'),
            hoverinfo='none',
            mode='lines')
```

```
        node_x = []
        node_y = []
        node_text = []
        node_color = []

        for node in G.nodes():
            x, y = pos[node]
            node_x.append(x)
            node_y.append(y)
            node_text.append(f"Para {node}: {G.nodes[node]['content']}")
            node_color.append(G.nodes[node].get('conflict_score', 0))

        node_trace = go.Scatter(
            x=node_x, y=node_y,
            mode='markers',
            hoverinfo='text',
            text=node_text,
            marker=dict(
                showscale=True,
                colorscale='YlOrRd',
                color=node_color,
                size=10,
                colorbar=dict(
                    thickness=15,
                    title='Conflict Score',
                    xanchor='left',
                    titleside='right'
                ),
                line_width=2))

        fig = go.Figure(data=[edge_trace, node_trace],
                 layout=go.Layout(
                    title='Paragraph Conflict Network',
                    titlefont_size=16,
                    showlegend=False,
                    hovermode='closest',
                    margin=dict(b=20,l=5,r=5,t=40),
                    xaxis=dict(showgrid=False, zeroline=False,
showticklabels=False),
```

```
                    yaxis=dict(showgrid=False, zeroline=False,
showticklabels=False))
                    )

        return fig

    def generate_conflict_report(self):
        """Generate a comprehensive report of conflicts and loopholes."""
        print("Generating comprehensive conflict report...")

        # Filter high-risk paragraphs
        high_risk = [p for p in self.paragraphs if p['conflict_score'] >
5]

        # Create a report dataframe
        report_data = []
        for i, para in enumerate(self.paragraphs):
            report_data.append({
                'Paragraph': i,
                'Section': para['section'],
                'Content': para['content'][:100] + "..." if
len(para['content']) > 100 else para['content'],
                'Conflict Score': para['conflict_score'],
                'Loopholes': para.get('loophole_count', 0),
                'Related Conflicts': len(para.get('conflicts', [])),
                'Compliance Issues': len(para.get('compliance_conflicts',
[]))
            })

        report_df = pd.DataFrame(report_data)
        report_df = report_df.sort_values('Conflict Score',
ascending=False)

        # Generate detailed statistics
        stats = {
            'total_paragraphs': len(self.paragraphs),
            'high_risk_paragraphs': len(high_risk),
            'paragraphs_with_loopholes': sum(1 for p in self.paragraphs if
'loophole_count' in p and p['loophole_count'] > 0),
```

```
            'paragraphs_with_compliance_issues': sum(1 for p in
self.paragraphs if 'compliance_conflicts' in p),
            'avg_conflict_score': sum(p['conflict_score'] for p in
self.paragraphs) / len(self.paragraphs),
            'max_conflict_score': max(p['conflict_score'] for p in
self.paragraphs),
            'most_conflicted_section': report_df.iloc[0]['Section'] if not
report_df.empty else "N/A"
        }

        return report_df, stats

    def highlight_document_conflicts(self):
        """Generate HTML with highlighted conflicts and loopholes."""
        highlighted_text = self.document_text

        # Create a color-coded HTML version
        html_parts = ["<h2>Document with Highlighted Conflicts</h2>"]
        html_parts.append("<div style='font-family: Arial; line-height:
1.6;'>")

        # Process each paragraph
        current_pos = 0
        for para in self.paragraphs:
            # Find the paragraph in the original text
            para_text = para['content']
            para_pos = highlighted_text.find(para_text, current_pos)

            if para_pos >= 0:
                # Add text before this paragraph
                html_parts.append(highlighted_text[current_pos:para_pos])

                # Determine paragraph color based on conflict score
                if para['conflict_score'] > 7:
                    bg_color = '#ffcccc'  # Red for high conflict
                elif para['conflict_score'] > 4:
                    bg_color = '#ffffcc'  # Yellow for medium conflict
                else:
                    bg_color = 'transparent'  # No highlight for low
conflict
```

```
                # Add the paragraph with appropriate highlighting
                html_parts.append(f"<p style='background-color:
{bg_color}; padding: 5px; margin: 5px 0;'>")
                html_parts.append(para_text)
                html_parts.append("</p>")

                current_pos = para_pos + len(para_text)

        # Add any remaining text
        html_parts.append(highlighted_text[current_pos:])
        html_parts.append("</div>")

        return "".join(html_parts)

# Example usage function
def run_legal_analyzer():
    """Example usage of the Legal Compliance Analyzer."""
    analyzer = LegalComplianceAnalyzer()

    # Option to upload or use sample text
    from google.colab import files
    import io

    print("Please select an option:")
    print("1. Upload a legal document file")
    print("2. Use sample legal text")

    option = input("Enter option (1 or 2): ")

    if option == "1":
        print("Please upload a text file containing the legal
document...")
        uploaded = files.upload()
        file_name = list(uploaded.keys())[0]
        file_content = uploaded[file_name].decode('utf-8')
        analyzer.load_document(text=file_content)
    else:
        # Sample legal text with intentional conflicts and loopholes
        sample_text = """
```

CONTRACT AGREEMENT

1. DEFINITIONS:

1.1 "Service Provider" means the party providing services under this Agreement.

1.2 "Client" means the recipient of services under this Agreement.

1.3 "Services" means the work performed by Service Provider as described in Section 2.

2. SCOPE OF SERVICES:

2.1 Service Provider shall provide the following services to Client: consulting, advisory, and implementation services related to data management.

2.2 Service Provider shall use reasonable efforts to meet all deadlines agreed upon by the parties.

2.3 Service Provider is not obligated to perform services outside the scope defined herein unless agreed in writing.

3. PAYMENT TERMS:

3.1 Client shall pay Service Provider at the rate of $X per hour for services rendered.

3.2 Payment is due within 30 days of invoice date.

3.3 Notwithstanding Section 3.2, payment for urgent services shall be due immediately upon completion.

3.4 Interest shall accrue on late payments at 1.5% per month or the maximum allowed by law, whichever is less.

4. CONFIDENTIALITY:

4.1 All information shared between parties shall be considered confidential.

4.2 Neither party shall disclose confidential information to third parties.

4.3 The confidentiality obligations shall not apply to information that is public knowledge.

4.4 Despite Section 4.1, Service Provider may, at its sole discretion, use Client's name for marketing purposes.

5. INTELLECTUAL PROPERTY:

5.1 All intellectual property created by Service Provider shall belong to Service Provider.

5.2 Client shall receive a non-exclusive license to use such intellectual property for business purposes.

5.3 Except as provided in Section 5.2, Client shall have no rights to Service Provider's intellectual property.

6. TERM AND TERMINATION:

6.1 This Agreement shall remain in effect until terminated by either party.

6.2 Either party may terminate this Agreement with 30 days written notice.

6.3 Client may terminate immediately in case of material breach by Service Provider.

6.4 Service Provider may terminate immediately if Client fails to make payments when due.

7. LIABILITY:

7.1 Service Provider's liability shall be limited to the amount paid by Client in the 3 months preceding any claim.

7.2 In no event shall Service Provider be liable for indirect or consequential damages.

7.3 Unless prohibited by applicable law, any claim must be brought within 6 months of the event giving rise to such claim.

8. GOVERNING LAW:

8.1 This Agreement shall be governed by the laws of the State of XYZ.

8.2 Any disputes shall be resolved by arbitration in the State of XYZ.

8.3 Notwithstanding Section 8.2, either party may seek injunctive relief in any court of competent jurisdiction.

9. MISCELLANEOUS:

9.1 This Agreement constitutes the entire understanding between the parties.

9.2 No amendment shall be effective unless in writing and signed by both parties.

9.3 If any provision is found to be unenforceable, the remaining provisions shall remain in effect.

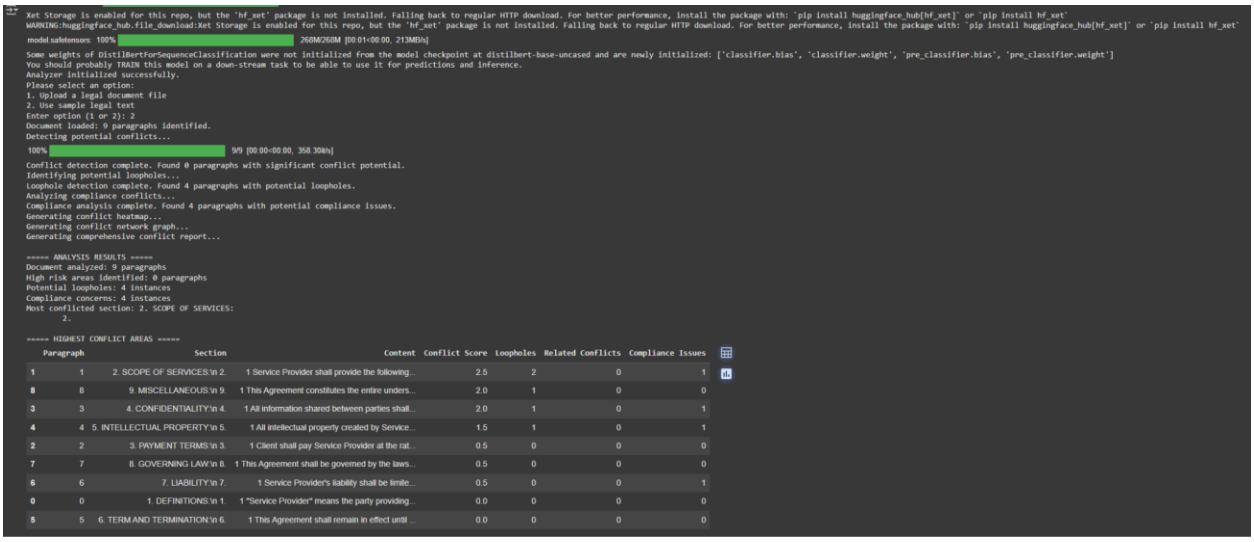
9.4 Neither party shall be liable for delays caused by circumstances beyond their reasonable control.

```
        9.5 Service Provider may assign this Agreement at its discretion.
Client may not assign without written consent.
        """
        analyzer.load_document(text=sample_text)

    # Run the analysis
    analyzer.detect_potential_conflicts()
    analyzer.identify_loopholes()

    # Sample regulations (in a real implementation, this would be more
comprehensive)
    sample_regulations = [
        {
            'name': 'Data Protection Regulation',
            'keywords': ['confidential', 'personal', 'data', 'privacy',
'disclosure']
        },
        {
            'name': 'Consumer Protection Law',
            'keywords': ['warranty', 'liability', 'damages', 'claim',
'termination']
        },
        {
            'name': 'Intellectual Property Rights',
            'keywords': ['intellectual property', 'license', 'copyright',
'trademark', 'patent']
        }
    ]

    analyzer.analyze_compliance_conflicts(regulations=sample_regulations)

    # Generate visualizations
    heatmap_fig, para_heatmap_fig = analyzer.generate_conflict_heatmap()
    network_fig = analyzer.generate_network_graph()

    # Generate report
    report_df, stats = analyzer.generate_conflict_report()

    # Display results
    print("\n===== ANALYSIS RESULTS =====")
```

```
    print(f"Document analyzed: {stats['total_paragraphs']} paragraphs")
    print(f"High risk areas identified: {stats['high_risk_paragraphs']}
paragraphs")
    print(f"Potential loopholes: {stats['paragraphs_with_loopholes']}
instances")
    print(f"Compliance concerns:
{stats['paragraphs_with_compliance_issues']} instances")
    print(f"Most conflicted section: {stats['most_conflicted_section']}")
    print("\n===== HIGHEST CONFLICT AREAS =====")
    display(report_df.head(10))

    # Display heatmap
    print("\n===== CONFLICT HEATMAP =====")
    heatmap_fig.show()
    para_heatmap_fig.show()

    # Display network graph
    print("\n===== CONFLICT NETWORK =====")
    network_fig.show()

    # Display highlighted document
    from IPython.display import HTML
    print("\n===== HIGHLIGHTED DOCUMENT =====")
    display(HTML(analyzer.highlight_document_conflicts()))

    return analyzer

# Run the analyzer when the script is executed
if __name__ == "__main__":
    print("Starting Legal Compliance Conflict Detection Tool...")
    analyzer = run_legal_analyzer()
```

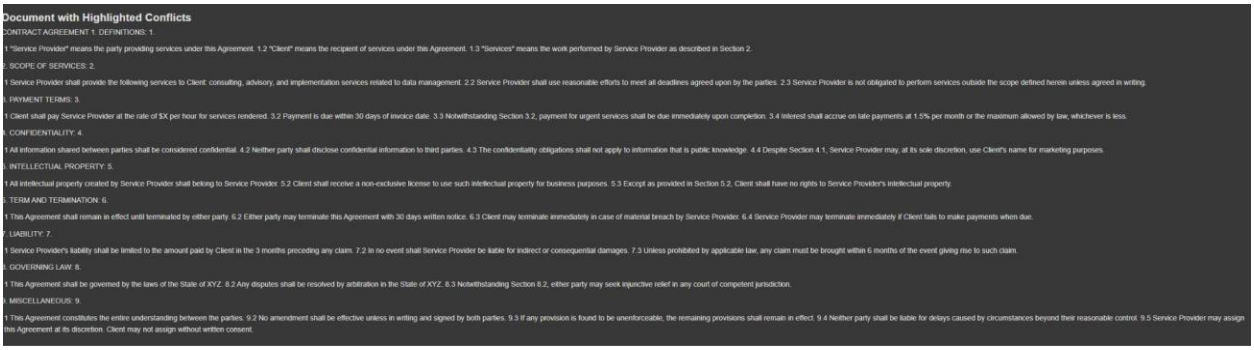
Output

The heatmap and conflict network arent generated as it is a sample text, an external file will produce both items.



```
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better performance, install the package with: `pip install huggingface_hub[hf_xet]` or `pip install hf_xet`
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better performance, install the package with: `pip install huggingface_hub[hf_xet]` or `pip install hf_xet`
model.safetensors: 100% 268M/268M [00:01<00:00, 213MB/s]
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight', 'pre_classifier.bias', 'pre_classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Analyzer initialized successfully.
Please select an option:
1. Upload a legal document file
2. Use sample legal text
Enter option (1 or 2): 2
Document loaded: 9 paragraphs identified.
Detecting potential conflicts...
100% 9/9 [00:00<00:00, 358.30it/s]
Conflict detection complete. Found 0 paragraphs with significant conflict potential.
Identifying potential loopholes...
Loophole detection complete. Found 4 paragraphs with potential loopholes.
Analyzing compliance conflicts...
Compliance analysis complete. Found 4 paragraphs with potential compliance issues.
Generating conflict heatmap...
Generating conflict network graph...
Generating comprehensive conflict report...

===== ANALYSIS RESULTS =====
Document analyzed: 9 paragraphs
High risk areas identified: 0 paragraphs
Potential loopholes: 4 instances
Compliance concerns: 4 instances
Most conflicted section: 2. SCOPE OF SERVICES:
        2.

===== HIGHEST CONFLICT AREAS =====
```

	Paragraph	Section	Content	Conflict Score	Loopholes	Related Conflicts	Compliance Issues
1	1	2. SCOPE OF SERVICES:\n 2.	1 Service Provider shall provide the following...	2.5	2	0	1
8	8	9. MISCELLANEOUS:\n 9.	1 This Agreement constitutes the entire unders...	2.0	1	0	0
3	3	4. CONFIDENTIALITY:\n 4.	1 All information shared between parties shall...	2.0	1	0	1
4	4	5. INTELLECTUAL PROPERTY:\n 5.	1 All intellectual property created by Service...	1.5	1	0	1
2	2	3. PAYMENT TERMS:\n 3.	1 Client shall pay Service Provider at the rat...	0.5	0	0	0
7	7	8. GOVERNING LAW:\n 8.	1 This Agreement shall be governed by the laws...	0.5	0	0	0
6	6	7. LIABILITY:\n 7.	1 Service Provider's liability shall be limite...	0.5	0	0	1
0	0	1. DEFINITIONS:\n 1.	1 "Service Provider" means the party providing...	0.0	0	0	0
5	5	6. TERM AND TERMINATION:\n 6.	1 This Agreement shall remain in effect until ...	0.0	0	0	0



Document with Highlighted Conflicts

CONTRACT AGREEMENT 1. DEFINITIONS: 1.

1 "Service Provider" means the party providing services under this Agreement. 1.2 "Client" means the recipient of services under this Agreement. 1.3 "Services" means the work performed by Service Provider as described in Section 2.

2. SCOPE OF SERVICES: 2.

1 Service Provider shall provide the following services to Client: consulting, advisory, and implementation services related to data management. 2.2 Service Provider shall use reasonable efforts to meet all deadlines agreed upon by the parties. 2.3 Service Provider is not obligated to perform services outside the scope defined herein unless agreed in writing.

3. PAYMENT TERMS: 3.

1 Client shall pay Service Provider at the rate of $X per hour for services rendered. 3.2 Payment is due within 30 days of invoice date. 3.3 Notwithstanding Section 3.2, payment for urgent services shall be due immediately upon completion. 3.4 Interest shall accrue on late payments at 1.5% per month or the maximum allowed by law, whichever is less.

4. CONFIDENTIALITY: 4.

1 All information shared between parties shall be considered confidential. 4.2 Neither party shall disclose confidential information to third parties. 4.3 The confidentiality obligations shall not apply to information that is public knowledge. 4.4 Despite Section 4.1, Service Provider may, at its sole discretion, use Client's name for marketing purposes.

5. INTELLECTUAL PROPERTY: 5.

1 All intellectual property created by Service Provider shall belong to Service Provider. 5.2 Client shall receive a non-exclusive license to use such intellectual property for business purposes. 5.3 Except as provided in Section 5.2, Client shall have no rights to Service Provider's intellectual property.

6. TERM AND TERMINATION: 6.

1 This Agreement shall remain in effect until terminated by either party. 6.2 Either party may terminate this Agreement with 30 days written notice. 6.3 Client may terminate immediately in case of material breach by Service Provider. 6.4 Service Provider may terminate immediately if Client fails to make payments when due.

7. LIABILITY: 7.

1 Service Provider's liability shall be limited to the amount paid by Client in the 3 months preceding any claim. 7.2 In no event shall Service Provider be liable for indirect or consequential damages. 7.3 Unless prohibited by applicable law, any claim must be brought within 6 months of the event giving rise to such claim.

8. GOVERNING LAW: 8.

1 This Agreement shall be governed by the laws of the State of XYZ. 8.2 Any disputes shall be resolved by arbitration in the State of XYZ. 8.3 Notwithstanding Section 8.2, either party may seek injunctive relief in any court of competent jurisdiction.

9. MISCELLANEOUS: 9.

1 This Agreement constitutes the entire understanding between the parties. 9.2 No amendment shall be effective unless in writing and signed by both parties. 9.3 If any provision is found to be unenforceable, the remaining provisions shall remain in effect. 9.4 Neither party shall be liable for delays caused by circumstances beyond their reasonable control. 9.5 Service Provider may assign this Agreement at its discretion. Client may not assign without written consent.

4. Chris To create a time-aware bias detection and political bias framing analysis

Report

Task

To create a time-aware bias detection and political bias framing analysis

PROGRAMS/TOOLS USED

1. Stack overflow
2. Kaggle
3. Google
4. Google colab

STEP-BY-STEP BREAKDOWN

1. First install the requirements: This includes the “vaderSentiment” which is an analysis tool. “Nltk” which is a toolkit used for processing the text, and finally “nltk.download('stopwords')” which essentially removes common unnecessary words which will be removed in the preprocessing stage.

2. Import the necessary libraries: The libraries installed are “pandas” which is a library for handling data frames. “Re” is used for clearing, “matplotlib” and “seaborn” are used for plotting the graph. “Stopwords” library will be used (imported) into “nltk” which will be used to filter the unnecessary words.

3. Use sample data: Used a small dataset with dates and text for testing purposes, this is

4. Clean and process the text, for example removing stopwords

5. Categorise keyword variables into a function. What this does is aid in building "frames," or collections of words that fall into particular categories. Ultimately what happens is the function examines each after preprocessing text to see if any of the categories' frames) keywords are present.

6. Add the rows and columns were necessary to group the frames and results

Code

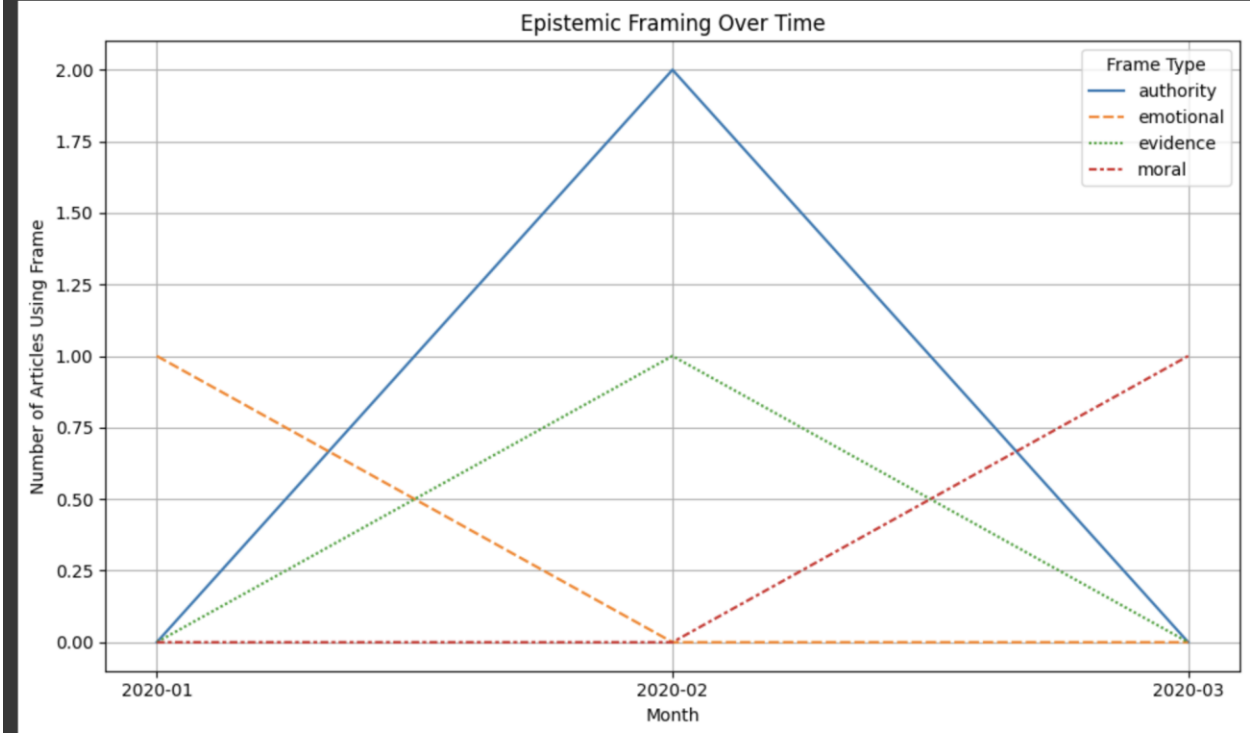
```
# ☑ STEP 0: Install requirements (if not already installed)
!pip install vaderSentiment --quiet
import nltk
nltk.download('stopwords')

# ☑ STEP 1: Import Libraries
import pandas as pd
import re
import matplotlib.pyplot as plt
import seaborn as sns
from nltk.corpus import stopwords
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

# ☑ STEP 2: Create Sample Data (replace this with your own CSV if needed)
data = {
    'date': ['2020-01-15', '2020-02-10', '2020-02-20', '2020-03-05'],
    'text': [
        "We must act now — lives are at stake! The situation is dire.",
        "According to recent studies, carbon levels have increased
dramatically.",
        "Experts from the World Health Organization advise immediate
action.",
        "It's our moral duty to protect the planet for future generations."
    ]
}
df = pd.DataFrame(data)

# ☑ STEP 3: Clean and preprocess text
stop_words = set(stopwords.words('english'))

def preprocess(text):
    text = text.lower()
    text = re.sub(r'\W+', ' ', text)
    tokens = text.split()
    tokens = [word for word in tokens if word not in stop_words]
    return ' '.join(tokens)
```

```
df['clean_text'] = df['text'].apply(preprocess)

# ☑ STEP 4: Convert dates and extract time period
df['date'] = pd.to_datetime(df['date'])
df['month'] = df['date'].dt.to_period('M')

# ☑ STEP 5: Define framing categories using keyword sets
framing_keywords = {
   'emotional': ['fear', 'urgent', 'crisis', 'outrage', 'dire',
'alarming', 'terrifying', 'horrific'],
   'evidence': ['data', 'study', 'studies', 'statistical', 'research',
'survey', 'evidence', 'graph', 'report'],
   'authority': ['experts', 'scientists', 'officials', 'world health
organization', 'dr', 'professor', 'government'],
   'moral': ['duty', 'responsibility', 'right', 'wrong', 'justice',
'ethical', 'moral', 'values'],
   'conflict': ['vs', 'against', 'battle', 'fight', 'war', 'clash',
'divide', 'opposition']
}

def detect_frames(text):
   frames = []
   for frame_type, keywords in framing_keywords.items():
       if any(word in text for word in keywords):
           frames.append(frame_type)
   return frames if frames else ['none']

df['frames'] = df['clean_text'].apply(detect_frames)

# ☑ STEP 6: Expand the 'frames' list into individual rows
df_exploded = df.explode('frames')

# ☑ STEP 7: Group and count frame usage by month
frame_counts = df_exploded.groupby(['month',
'frames']).size().unstack(fill_value=0)
frame_counts.index = frame_counts.index.astype(str)
```

```
# ☑ STEP 8: Plot epistemic frame usage over time
plt.figure(figsize=(10, 6))
sns.lineplot(data=frame_counts)
plt.title("Epistemic Framing Over Time")
plt.xlabel("Month")
plt.ylabel("Number of Articles Using Frame")
plt.grid(True)
plt.legend(title="Frame Type")
plt.tight_layout()
plt.show()

# ☑ Optional: Show final data
df[['date', 'text', 'frames']]
```

Output

```
126.0/126.0 kB 5.4 MB/s eta 0:00:00
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```



1 to 4 of 4 entries Filter

index	date	text	frames
0	2020-01-15 00:00:00	We must act now — lives are at stake! The situation is dire.	emotional
1	2020-02-10 00:00:00	According to recent studies, carbon levels have increased dramatically.	evidence,authority
2	2020-02-20 00:00:00	Experts from the World Health Organization advise immediate action.	authority
3	2020-03-05 00:00:00	It's our moral duty to protect the planet for future generations.	moral

5. Leon Adjustable trust thresholds and pre-trained domain-specific trust profiles

This is the html code to do this; it's for a website login. This is pseudo-code as to actually test and run this would require building a fully functioning website.

This is a python code to evaluate it on a computer. Yet again, it's pseudo-code as actually testing it would require it be installed and run on separate individual PCs. This is however a functioning code.

Report

Task

To create a tool that has Adjustable trust thresholds and pre-trained domain-specific trust profiles

PROGRAMS/TOOLS USED

1. Stack overflow
2. Google
3. Google colab

Step by step

Googled the definition and meaning of the subject.
Youtubed and looked got chatgpt to aid assist me when i got stuck, code works, but it is pseudo-code, so it will require the installation of this on individual computers just to test it.
Asked ChatGPT to generate a code in html, but more specifically aimed at a website and for the login function.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Login</title>
  <style>
```

```
      body {
        font-family: Arial, sans-serif;
        background: #f5f5f5;
        display: flex;
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Login</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background: #f5f5f5;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }
    .login-container {
      background: white;
      padding: 2rem;
      border-radius: 10px;
      box-shadow: 0 4px 12px rgba(0, 0, 0, 0.1);
      width: 300px;
    }
    h2 {
      text-align: center;
      margin-bottom: 1.5rem;
    }
    label {
      display: block;
      margin-bottom: 0.5rem;
      margin-top: 1rem;
    }
    input[type="email"],
    input[type="password"] {
      width: 100%;
      padding: 0.5rem;
      margin-bottom: 0.5rem;
      border: 1px solid #ccc;
```

```
            border-radius: 5px;
          }
          button {
            width: 100%;
            padding: 0.75rem;
            background-color: #007bff;
            color: white;
            border: none;
            border-radius: 5px;
            font-size: 1rem;
            cursor: pointer;
            margin-top: 1rem;
          }
          button:hover {
            background-color: #0056b3;
          }
          .signup-link {
            text-align: center;
            margin-top: 1rem;
            font-size: 0.9rem;
          }
        </style>
      </head>
      <body>

        <div class="login-container">
          <h2>Login</h2>
          <form action="/login" method="POST">
            <label for="email">Email</label>
            <input type="email" id="email" name="email" required />

            <label for="password">Password</label>
            <input type="password" id="password" name="password" required
/>

            <button type="submit">Login</button>
          </form>
          <div class="signup-link">
            Don't have an account? <a href="/signup">Sign up</a>
          </div>
```

```
    </div>

  </body>
  </html>
        background: white;
        padding: 2rem;
        border-radius: 10px;
        box-shadow: 0 4px 12px rgba(0, 0, 0, 0.1);
        width: 300px;
      }
      h2 {
        text-align: center;
        margin-bottom: 1.5rem;
      }
      label {
        display: block;
        margin-bottom: 0.5rem;
        margin-top: 1rem;
      }
      input[type="email"],
      input[type="password"] {
        width: 100%;
        padding: 0.5rem;
        margin-bottom: 0.5rem;
        border: 1px solid #ccc;
        border-radius: 5px;
      }
      button {
        width: 100%;
        padding: 0.75rem;
        background-color: #007bff;
        color: white;
        border: none;
        border-radius: 5px;
        font-size: 1rem;
        cursor: pointer;
        margin-top: 1rem;
      }
      button:hover {
        background-color: #0056b3;
```

```
          }
          .signup-link {
            text-align: center;
            margin-top: 1rem;
            font-size: 0.9rem;
          }
        </style>
      </head>
      <body>

        <div class="login-container">
          <h2>Login</h2>
          <form action="/login" method="POST">
            <label for="email">Email</label>
            <input type="email" id="email" name="email" required />

            <label for="password">Password</label>
            <input type="password" id="password" name="password" required
/>

            <button type="submit">Login</button>
          </form>
          <div class="signup-link">
            Don't have an account? <a href="/signup">Sign up</a>
          </div>
        </div>

      </body>
      </html>
```

```
# Define a domain-specific trust profile
class TrustProfile:
    def __init__(self, domain_name, trust_factors):
        """
        :param domain_name: Name of the domain (e.g., 'IoT', 'Healthcare')
        :param trust_factors: Dict of factor: weight (e.g., {'uptime': 0.4, 'firmware_integrity': 0.6})
        """
        self.domain = domain_name
        self.trust_factors = trust_factors  # Pre-trained weights for each trust factor
```

```
    def evaluate(self, metrics):
        """
        Evaluate trust score using metrics (e.g., actual device values)
        :param metrics: Dict of trust factor: score (e.g., {'uptime': 0.9, 'firmware_integrity': 0.8})
        """
        score = 0.0
        for factor, weight in self.trust_factors.items():
            score += metrics.get(factor, 0) * weight
        return round(score, 3)

# Define the evaluator with adjustable thresholds
class TrustEvaluator:
    def __init__(self, threshold=0.75):
        """
        :param threshold: Trust score threshold (can be adjusted)
        """
        self.threshold = threshold

    def set_threshold(self, new_threshold):
        self.threshold = new_threshold

    def is_trustworthy(self, score):
        return score >= self.threshold

# Example usage
if __name__ == "__main__":
    # Pre-trained profile for IoT devices
    iot_profile = TrustProfile(
        domain_name="IoT",
        trust_factors={
            'uptime': 0.3,
            'firmware_integrity': 0.4,
            'anomaly_detection_score': 0.3
        }
    )

    # Adjustable threshold evaluator
    evaluator = TrustEvaluator(threshold=0.8)

    # Sample metrics from a device
    device_metrics = {
        'uptime': 0.95,
        'firmware_integrity': 0.85,
        'anomaly_detection_score': 0.6
    }
```

```
trust_score = iot_profile.evaluate(device_metrics)
print(f"Trust Score: {trust_score}")

if evaluator.is_trustworthy(trust_score):
    print("Device is TRUSTWORTHY ✅")
else:
    print("Device is UNTRUSTWORTHY ⚠️")
```

6. Daryl Create a fact-check api integration, a custom alert system for contradiction spikes

Report

Task

Create a fact-check api integration, a custom alert system for contradiction spikes

Step By Step:

Dependency installation: The code starts by importing all required Python libraries which include Flask for the web server, Pyngrok for tunneling, SQLAlchemy for database ORM, Requests for API communication, Flask-CORS for handling cross-origin requests and NLTK for basic text processing.

Library imports and NLTK setup: It then imports essential modules and downloads NLTK datasets (punkt and stopwords) needed for text tokenization and filtering.

Flask app initialization and CORS configuration: A Flask application is initialized and CORS is enabled to make possible cross origin requests from the frontend interface.

Database setup with SQLAlchemy: The app configures an SQLite database and defines a FactCheckResult model to store user claims, fact-check ratings, sources, URLs, and timestamps. The necessary tables are created automatically within the application context.

Google API key retrieval: It tries to securely load the Google Fact Check API key from either Google Colab or environment variables, depending on the execution environment.

HTML frontend definition: A styled HTML interface is provided as a multiline string. It includes a text input for claims, a submission button, and dynamically updated result sections with alerts styled by the claim outcome.

Fallback knowledge base: KNOWLEDGE_BASE is a simple dictionary that serves as a backup source of truth for certain hard-coded claims that will be used when the Google Fact Check API is unavailable or when the Google Fact Check API does not return any result.

Fallback logic via basic_fact_check(): This function performs a basic keyword match against the fallback knowledge base and returns a pre-formulated assessment of the claim's accuracy.

Homepage routing: The root endpoint (/) renders the HTML interface using Flask's render_template_string() function.

Fact-check API endpoint: A POST endpoint (/fact-check) handles incoming claims, queries the Google Fact Check API, stores the response in the database, and returns either API results or fallback responses depending on availability or errors.

Contradiction alert system: The /alerts endpoint looks at the last 10 fact-check results for contradiction-related ratings such as "false", "pants on fire". It then returns an alert level (Low, Medium, or High) along with summary data on the contradictory claims.

Ngrok tunnel initialization: The code attempts to open a public ngrok tunnel so the app can be accessed remotely, and prints the generated public URL.

App execution: Finally, the Flask server is started on all available network interfaces (0.0.0.0) at port 5000, enabling the app to run locally or via ngrok.

Code

```
!pip install flask pyngrok flask-sqlalchemy requests flask-cors nltk --
quiet
from flask import Flask, request, jsonify, render_template_string
from flask_sqlalchemy import SQLAlchemy
from flask_cors import CORS
from pyngrok import ngrok
import requests
from datetime import datetime
import os
import json
import nltk
import re

# Download NLTK data
nltk.download('punkt', quiet=True)
nltk.download('stopwords', quiet=True)

# Initialize Flask
app = Flask(__name__)
CORS(app)  # Add CORS support for frontend requests
```

```
# Configure SQLite database
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///factchecks.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

# Try to get Google API key securely
try:
    from google.colab import userdata
    GOOGLE_API_KEY = userdata.get('GOOGLE_API_KEY')
except:
    # Fallback if not in Colab or key not set
    GOOGLE_API_KEY = os.environ.get('GOOGLE_API_KEY')

# Database Model
class FactCheckResult(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    claim = db.Column(db.String(500), nullable=False)
    rating = db.Column(db.String(50))
    source = db.Column(db.String(100))
    url = db.Column(db.String(500))
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)

# Create tables
with app.app_context():
    db.create_all()

# Simple HTML frontend with improved styling
HOMEPAGE = '''
<!DOCTYPE html>
<html>
<head>
    <title>Fact Check API</title>
    <style>
        body { font-family: Arial, sans-serif; max-width: 800px; margin: 0
auto; padding: 20px; background-color: #f9f9f9; }
        h1 { color: #2c3e50; text-align: center; }
        .form-group { margin-bottom: 15px; }
        label { display: block; margin-bottom: 5px; font-weight: bold; }
        input[type="text"] { width: 100%; padding: 12px; font-size: 16px;
border: 1px solid #ddd; border-radius: 4px; box-sizing: border-box; }
```

```
        button { background: #3498db; color: white; border: none; padding:
12px 20px; cursor: pointer; width: 100%; font-size: 16px; border-radius:
4px; }
        button:hover { background: #2980b9; }
        #result { margin-top: 20px; border: 1px solid #ddd; padding: 15px;
border-radius: 4px; white-space: pre-wrap; background-color: white; }
        .loading { color: #7f8c8d; text-align: center; }
        .alert { padding: 10px; margin-bottom: 15px; border-radius: 4px; }
        .alert-info { background-color: #d1ecf1; border: 1px solid #bee5eb;
color: #0c5460; }
        .alert-success { background-color: #d4edda; border: 1px solid
#c3e6cb; color: #155724; }
        .alert-warning { background-color: #fff3cd; border: 1px solid
#ffeeba; color: #856404; }
        .alert-danger { background-color: #f8d7da; border: 1px solid
#f5c6cb; color: #721c24; }
    </style>
</head>
<body>
    <h1>Fact Check API</h1>
    <div class="alert alert-info">
        Enter a claim below to check its factual accuracy. Our system will
analyze the claim and provide results.
    </div>
    <div class="form-group">
        <label for="claim">Enter a claim to fact check:</label>
        <input type="text" id="claim" placeholder="e.g., The earth is
flat">
    </div>
    <button onclick="checkFact()">Check Fact</button>
    <div id="result"></div>

    <script>
        async function checkFact() {
            const claim = document.getElementById('claim').value;
            if (!claim) return;

            const resultDiv = document.getElementById('result');
            resultDiv.innerHTML = '<p class="loading">Processing your
request...</p>';
```

```
            try {
                const response = await fetch('/fact-check', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json',
                    },
                    body: JSON.stringify({ claim }),
                });

                const data = await response.json();

                // Format the response nicely
                let html = '<h3>Results:</h3>';

                if (data.results && data.results.length > 0) {
                    html += '<div class="alert alert-success">Fact check
completed successfully!</div>';
                    html += '<ul>';
                    data.results.forEach(result => {
                        html += `<li><strong>Rating:</strong>
${result.rating}<br>`;
                        html += `<strong>Source:</strong>
${result.source}<br>`;
                        if (result.url) html += `<a href="${result.url}"
target="_blank">Read more</a>`;
                        html += '</li>';
                    });
                    html += '</ul>';
                } else if (data.basic_check) {
                    const alertClass = data.basic_check.likely_true ?
'alert-success' : 'alert-danger';
                    html += `<div class="alert
${alertClass}">${data.basic_check.assessment}</div>`;
                    if (data.basic_check.explanation) {
                        html += `<p>${data.basic_check.explanation}</p>`;
                    }
                } else if (data.error) {
                    html += `<div class="alert alert-warning">We
encountered an issue: ${data.error}</div>`;
```

```
                    html += '<p>Using fallback fact checking method.</p>';
                }

                resultDiv.innerHTML = html;
            } catch (error) {
                resultDiv.innerHTML = `<div class="alert alert-
danger">Error: ${error.message}</div>`;
            }
        }
    </script>
</body>
</html>
'''

# Simple knowledge base for fallback fact checking
KNOWLEDGE_BASE = {
    "earth flat": {
        "likely_true": False,
        "assessment": "The claim that the Earth is flat is FALSE.",
        "explanation": "Scientific evidence overwhelmingly confirms that
the Earth is roughly spherical. This has been proven through satellite
imagery, circumnavigation, physics of gravity, and direct observation of
the Earth's curvature."
    },
    "climate change": {
        "likely_true": True,
        "assessment": "Climate change is real and primarily caused by human
activities.",
        "explanation": "Scientific consensus based on thousands of studies
shows that global warming and climate change are occurring and are
primarily caused by human activities, especially the burning of fossil
fuels."
    },
    "vaccines autism": {
        "likely_true": False,
        "assessment": "Vaccines do NOT cause autism.",
        "explanation": "Multiple large-scale studies have found no link
between vaccines and autism. The original study that suggested this
connection was retracted due to serious procedural errors and ethical
violations."
```

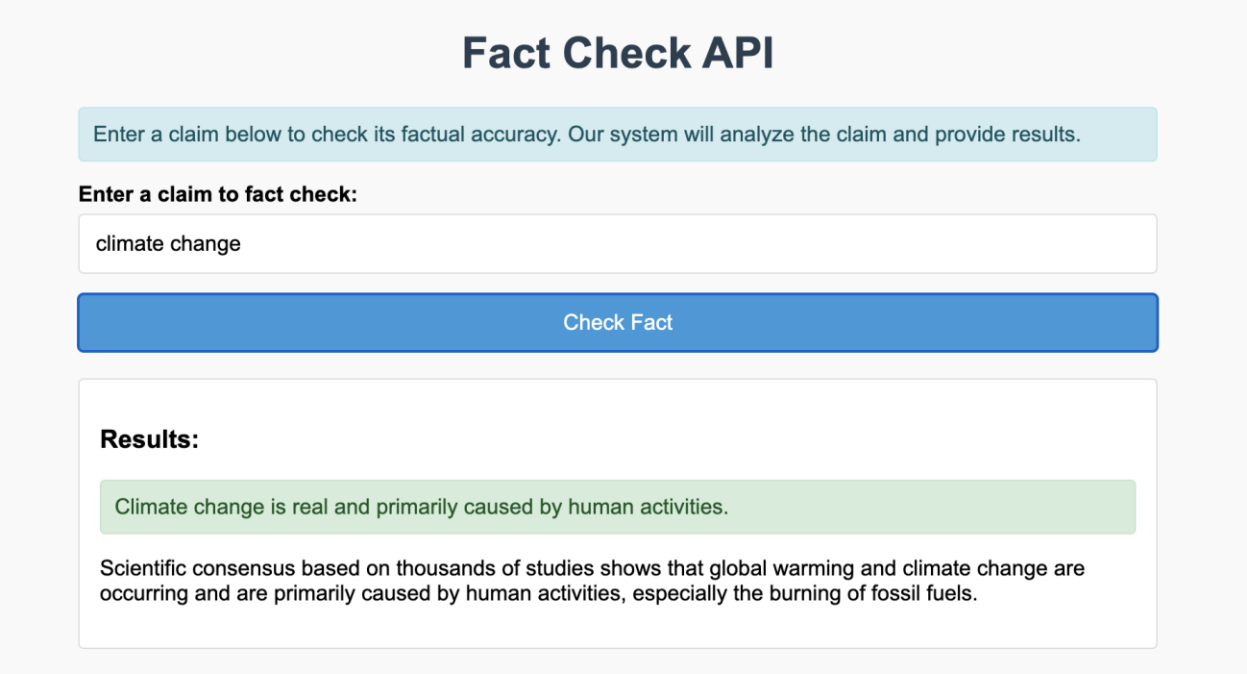
```
    }
}

def basic_fact_check(claim):
    """Simple keyword-based fact checking when API fails"""
    claim_lower = claim.lower()

    # Check against our simple knowledge base
    for key, info in KNOWLEDGE_BASE.items():
        if all(word in claim_lower for word in key.split()):
            return info

    # Default response if no match
    return {
        "likely_true": None,
        "assessment": "Unable to verify this specific claim with our
fallback system.",
        "explanation": "We couldn't determine the veracity of this claim
with our basic fact-checking system. Please check reliable sources for
more information."
    }

@app.route('/')
def home():
    return render_template_string(HOMEPAGE)

@app.route('/fact-check', methods=['POST'])
def check_claim():
    data = request.json if request.is_json else request.form.to_dict()
    claim = data.get('claim')

    if not claim:
        return jsonify({"error": "Missing claim"}), 400

    if not GOOGLE_API_KEY:
        # Use fallback if no API key
        basic_result = basic_fact_check(claim)
        return jsonify({
            "claim": claim,
            "message": "Claim processed with fallback system",
```

```
            "basic_check": basic_result
        })

    try:
        # Call Google Fact Check API
        params = {
            'query': claim,
            'key': GOOGLE_API_KEY,
            'languageCode': 'en'
        }

        response = requests.get(
            'https://factchecktools.googleapis.com/v1alpha1/claims:search',
            params=params,
            timeout=10  # Add timeout
        )
        response.raise_for_status()

        results = response.json().get('claims', [])
        processed_results = []

        # Process and store results
        for result in results:
            review = result.get('claimReview', [{}])[0]
            factcheck = FactCheckResult(
                claim=claim,
                rating=review.get('textualRating', 'No rating'),
                source=result.get('publisher', {}).get('name', 'Unknown'),
                url=review.get('url', '')
            )
            db.session.add(factcheck)

            processed_results.append({
                "rating": factcheck.rating,
                "source": factcheck.source,
                "url": factcheck.url
            })

        db.session.commit()
```

```
        if not processed_results:
            # No results from API, use fallback
            basic_result = basic_fact_check(claim)
            return jsonify({
                "claim": claim,
                "message": "No API results found, using fallback",
                "results": [],
                "basic_check": basic_result
            })

        return jsonify({
            "claim": claim,
            "results": processed_results[:3],  # Return top 3 results
            "total_found": len(results)
        })

    except Exception as e:
        # API call failed, use fallback
        basic_result = basic_fact_check(claim)
        return jsonify({
            "claim": claim,
            "message": "Claim received",
            "error": str(e),
            "basic_check": basic_result
        })

# Add alerts endpoint
@app.route('/alerts', methods=['GET'])
def get_alerts():
    # Check for contradiction spikes
    try:
        recent_checks =
FactCheckResult.query.order_by(FactCheckResult.timestamp.desc()).limit(10)
.all()
        contradictions = [check for check in recent_checks if check.rating
and
                          ("false" in check.rating.lower() or
                           "mostly false" in check.rating.lower() or
                           "pants on fire" in check.rating.lower())]
```

```
        alert_level = "Low"
        if len(contradictions) >= 7:
            alert_level = "High"
        elif len(contradictions) >= 4:
            alert_level = "Medium"

        return jsonify({
            "alert_level": alert_level,
            "contradictions_count": len(contradictions),
            "recent_checks": len(recent_checks),
            "recent_contradictions": [
                {"claim": c.claim, "rating": c.rating, "source": c.source}
                for c in contradictions[:3]  # Top 3 contradictions
            ]
        })
    except Exception as e:
        return jsonify({"error": str(e)}), 500

# Start ngrok tunnel
try:
    public_url = ngrok.connect(5000).public_url
    print(f"\n🔥 ACCESS YOUR FACT CHECK API HERE: {public_url}\n")
except Exception as e:
    print(f"Could not start ngrok: {str(e)}")
    public_url = "http://127.0.0.1:5000"

# Start server
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Output


Fact Check API
Enter a claim below to check its factual accuracy. Our system will analyze the claim and provide results.
Enter a claim to fact check:
climate change
Check Fact
Results:
Climate change is real and primarily caused by human activities.
Scientific consensus based on thousands of studies shows that global warming and climate change are occurring and are primarily caused by human activities, especially the burning of fossil fuels.

Fact Check API

Enter a claim below to check its factual accuracy. Our system will analyze the claim and provide results.

Enter a claim to fact check:

earth is flat

Check Fact

Results:

The claim that the Earth is flat is FALSE.

Scientific evidence overwhelmingly confirms that the Earth is roughly spherical. This has been proven through satellite imagery, circumnavigation, physics of gravity, and direct observation of the Earth's curvature.

Fact Check API

Enter a claim below to check its factual accuracy. Our system will analyze the claim and provide results.

Enter a claim to fact check:

vaccines autism

Check Fact

Results:

Vaccines do NOT cause autism.

Multiple large-scale studies have found no link between vaccines and autism. The original study that suggested this connection was retracted due to serious procedural errors and ethical violations.

7. Vanessa Create a chain of trust visualisation, custom regulatory report generator

Report

Task

To create a chain of trust visualisation, custom regulatory report generator

PROGRAMS/TOOLS USED

1. Stack overflow
2. Google
3. Google colab

STEP-BY-STEP BREAKDOWN

Step 1: Install Required Libraries, graphviz and reportlab are needed in order to create a visual representation of the report and diagram.
Step 2: Imports, allows for essential functions and libraries to be utilised for the end product.
Step 3: Chain of Trust Visualizer, this outlines how the chain of trust diagram will be made through graphviz.
Step 4: Custom Report Generator with Embedded Image, this step compiles the regulatory report.
Step 5: Interactive User Input, allows the user to input text regarding the report.

```
# ☑ Step 1: Install Required Libraries
!pip install graphviz reportlab --quiet

# ☑ Step 2: Imports
from graphviz import Digraph
from reportlab.lib.pagesizes import A4
from reportlab.pdfgen import canvas
from reportlab.lib.utils import ImageReader
import datetime
import os
from IPython.display import Image, display

# ☑ Step 3: Chain of Trust Visualizer
def generate_chain_of_trust(entities, output_name='chain_of_trust'):
    dot = Digraph(comment="Chain of Trust")
    dot.attr(rankdir='LR', size='8,5')

    for i, entity in enumerate(entities):
        dot.node(f"{i}", entity)

    for i in range(len(entities) - 1):
        dot.edge(f"{i}", f"{i + 1}")

    output_path = f"{output_name}.png"
    dot.render(filename=output_name, format='png', cleanup=True)
    return output_path

# ☑ Step 4: Custom Report Generator with Embedded Image
def generate_regulatory_report(title, findings, compliance_refs,
chain_img_path, output_format='pdf'):
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    filename = f"Regulatory_Report_{timestamp}"

    if output_format == 'pdf':
        filepath = f"{filename}.pdf"
        c = canvas.Canvas(filepath, pagesize=A4)
        width, height = A4
        y = height - 50
```

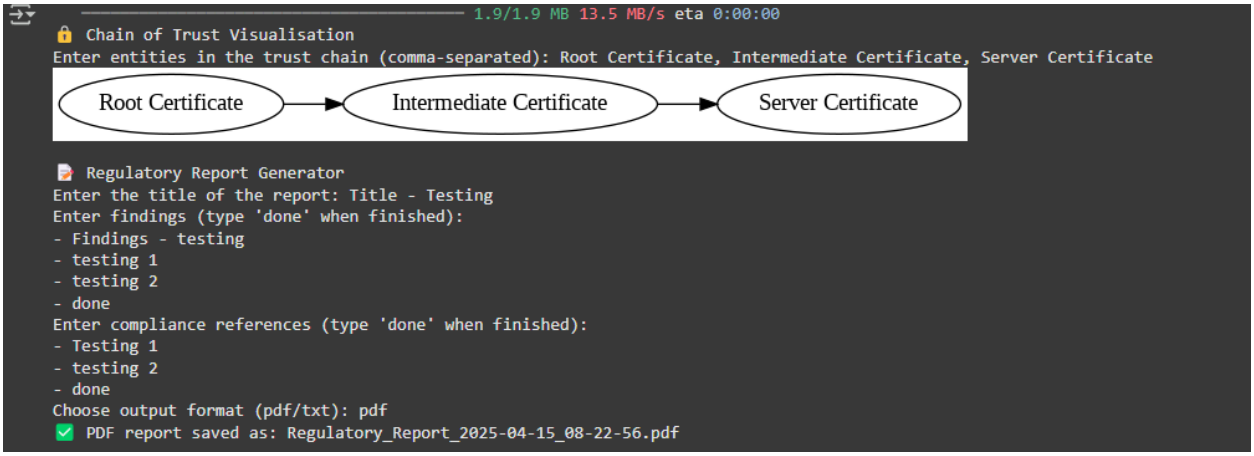
```
        # Add the image if it exists
        if os.path.exists(chain_img_path):
            image = ImageReader(chain_img_path)
            img_width = 500
            c.drawImage(image, 50, y - 220, width=img_width,
preserveAspectRatio=True, mask='auto')
            y -= 240  # Adjust for image height

        # Title
        c.setFont("Helvetica-Bold", 16)
        c.drawString(50, y, title)
        y -= 30

        # Date
        c.setFont("Helvetica", 12)
        c.drawString(50, y, f"Generated on: {datetime.datetime.now()}")
        y -= 40

        # Findings
        c.setFont("Helvetica-Bold", 14)
        c.drawString(50, y, "Findings:")
        y -= 20
        c.setFont("Helvetica", 12)
        for finding in findings:
            c.drawString(60, y, f"- {finding}")
            y -= 20
            if y < 100:
                c.showPage()
                y = height - 50

        # Compliance References
        y -= 20
        c.setFont("Helvetica-Bold", 14)
        c.drawString(50, y, "Compliance References:")
        y -= 20
        c.setFont("Helvetica", 12)
        for ref in compliance_refs:
            c.drawString(60, y, f"- {ref}")
            y -= 20
```

```
            if y < 100:
                c.showPage()
                y = height - 50

        c.save()
        print(f"☑ PDF report saved as: {filepath}")

    elif output_format == 'txt':
        filepath = f"{filename}.txt"
        with open(filepath, 'w') as f:
            f.write(f"{title}\n")
            f.write(f"Generated on: {datetime.datetime.now()}\n\n")
            f.write("[Diagram not included in text version]\n\n")
            f.write("Findings:\n")
            for finding in findings:
                f.write(f"- {finding}\n")
            f.write("\nCompliance References:\n")
            for ref in compliance_refs:
                f.write(f"- {ref}\n")
        print(f"☑ Text report saved as: {filepath}")

    else:
        print("⚠ Unsupported output format. Please use 'pdf' or 'txt'.")

# ☑ Step 5: Interactive User Input
def get_user_inputs():
    print("🔒 Chain of Trust Visualisation")
    entities = input("Enter entities in the trust chain (comma-separated):
").split(',')
    entities = [e.strip() for e in entities]
    image_path = generate_chain_of_trust(entities)
    if os.path.exists(image_path):
        display(Image(filename=image_path))
    else:
        print("⚠ Could not display diagram image.")

    print("\n📝 Regulatory Report Generator")
    title = input("Enter the title of the report: ")
```

```
    findings = []
    print("Enter findings (type 'done' when finished):")
    while True:
        f = input("- ")
        if f.lower() == 'done':
            break
        findings.append(f)

    compliance_refs = []
    print("Enter compliance references (type 'done' when finished):")
    while True:
        ref = input("- ")
        if ref.lower() == 'done':
            break
        compliance_refs.append(ref)

    output_format = input("Choose output format (pdf/txt): ").lower()
    if output_format not in ['pdf', 'txt']:
        output_format = 'pdf'
        print("⚠️ Invalid format. Defaulting to PDF.")

    generate_regulatory_report(title, findings, compliance_refs,
image_path, output_format)

# ☑️ Step 6: Run It
get_user_inputs()
```

Output

The output can be either in txt or pdf format



Title - Testing

Generated on: 2025-04-15 08:22:56.938206

Findings:

- Findings - testing
- testing 1
- testing 2

Compliance References:

- Testing 1
- testing 2