PolicyNav: Public Policy Navigation Using AI

Infosys Springboard Winter Internship 2025

Domain: Artificial Intelligence | Natural Language Processing

GitHub: **https://github.com/neharmenon05/infosys__springboard_internship**

Transforming Policy Access Through AI

The Challenge

Public policy documents are notoriously dense and complex. Citizens, researchers, and even policymakers struggle to navigate hundreds of pages to find relevant information. Manual searches are time-consuming and often miss critical context.

Our Solution

PolicyNav leverages advanced Natural Language Processing to analyze, summarize, and answer questions about policy documents in real-time. We've built an interactive FastAPI interface that understands plain language queries and returns precise policy insights.

AI-Powered Analysis

NLP models trained on education policy documents

Real-Time Q&A

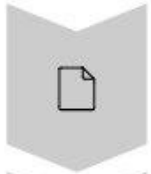
Interactive web interface for instant responses

Scalable Design

Modular architecture ready for health, finance domains

System Architecture Overview

Our end-to-end pipeline transforms raw policy documents into an intelligent, queryable knowledge base through six integrated phases.



Data Collection

Gathered government and public policy documents including education policy texts

Data Preprocessing

Clean, tokenize, and normalize text data using advanced NLP techniques

Model Training

Train NLP classifier and question-answering models on processed data

Model Serialization

Save trained model as .pkl file for efficient deployment

API Deployment

Serve predictions through FastAPI backend infrastructure

Frontend Interface

User-friendly web UI for seamless policy navigation

The AI Technology Stack

PolicyNav integrates multiple AI components to deliver comprehensive policy analysis capabilities. Each technology plays a strategic role in our system.

Machine Learning

Algorithms learn patterns from policy data to classify queries and identify relevant document sections

Deep Learning

Neural networks create sophisticated text representations using embeddings and transformers

Natural Language Processing

Core technology for query analysis, text extraction, and language understanding

Computer Vision

Extracts text from scanned policy documents and legacy formats

Generative AI

Creates concise policy summaries and synthesizes information from multiple sources

Agentic AI

Autonomous agents plan and execute complex policy retrieval and recommendation tasks

Core NLP Concepts Powering PolicyNav

Corpus

Our comprehensive collection of policy-related texts forms the foundation for model training and knowledge extraction

Tokenization

Breaking down policy documents into individual words and sentences for granular analysis

Stopwords Removal

Eliminating common words like "is," "and," and "the" to focus on meaningful content

Stemming & Lemmatization

Converting words to their base or root forms for consistent pattern recognition

POS Tagging

Identifying parts of speech—nouns, verbs, adjectives—to understand sentence structure

Named Entity Recognition

Extracting key entities such as laws, years, organizations, and policy names

Vectorization

Converting text into numerical features using TF-IDF and embeddings for machine learning

CLASSICAL METHOD

PART 1

PIPELINE OVERVIEW

1. Load Data

Import datasets from policy documents like the Primary Education Policy 2020

2. Preprocess Data

Apply tokenization, lemmatization, and stopword removal using NLTK, spaCy, and Hugging Face Transformers

3. Feature Engineering

Create TF-IDF vectors, Word2Vec, and BERT embeddings to convert text into machine-understandable formats

4. Training

Train classification and QA models using Logistic Regression, Naive Bayes, or fine-tuned BERT

5. Testing & Evaluation

Validate with accuracy, precision, recall, and F1 scores using sample queries

6. Deployment

Save model as .pkl file and integrate into FastAPI for real-time predictions

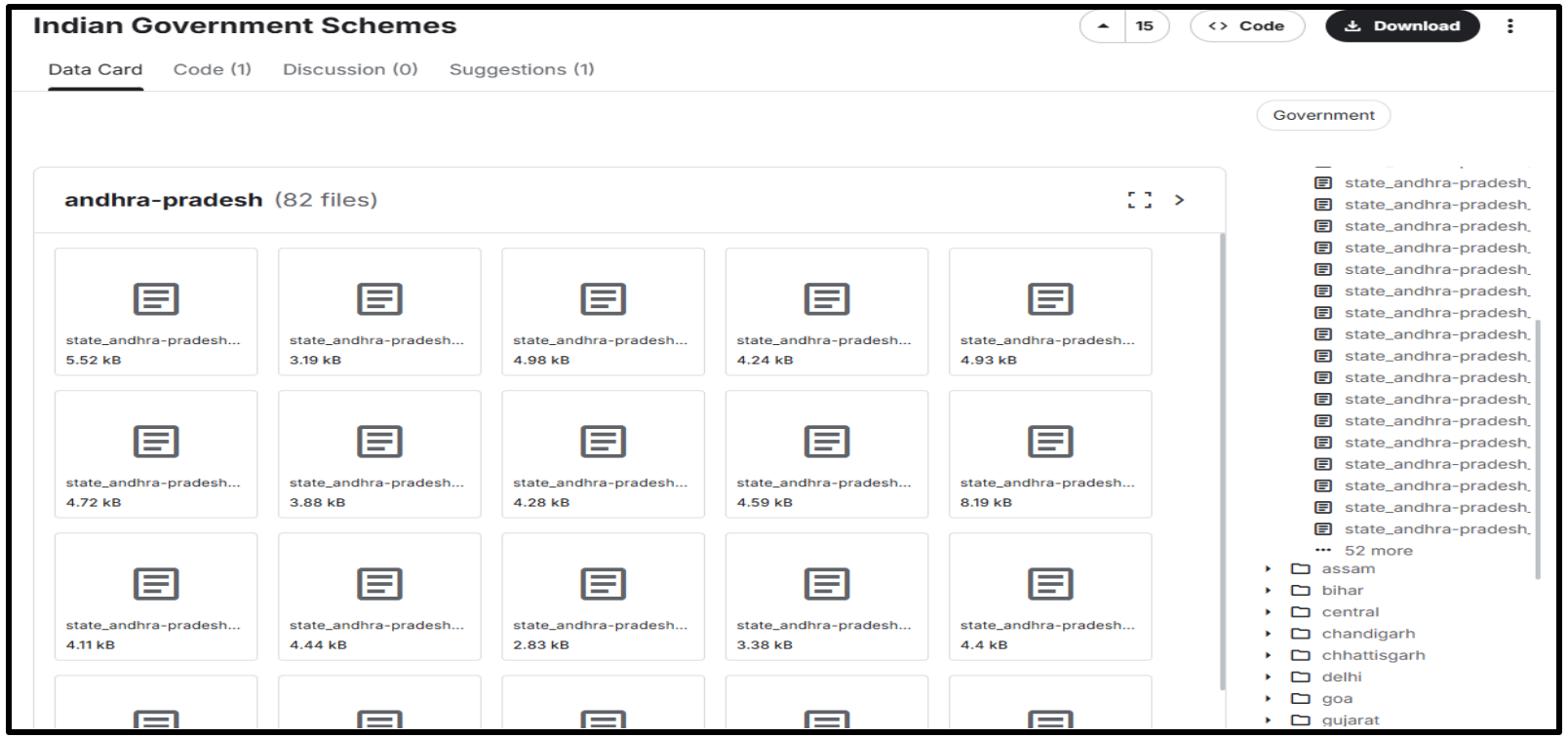
Load Data

1.1

DATASET OVERVIEW

Step 1: Collect DATA

https://www.kaggle.com/datasets/nitishabharathi/indian-government-schemes



Preprocess

Step 2: Create CSV and preprocess

Policy ID	Title	State	Year	Category	Status
1	Free Education for Girls	Tamil Nadu	2021	Education	Active

- `title` is taken from the **first line** of each `.txt` file.
- `state` is inferred from the folder name (e.g., `kaggle_dataset/tamil-nadu/...` → `Tamil Nadu`).
- `year` is extracted from the text
- `category` is assigned based on keyword rules in your function.
- `status` and `region`
- `"Active"` or "NOT" decided.

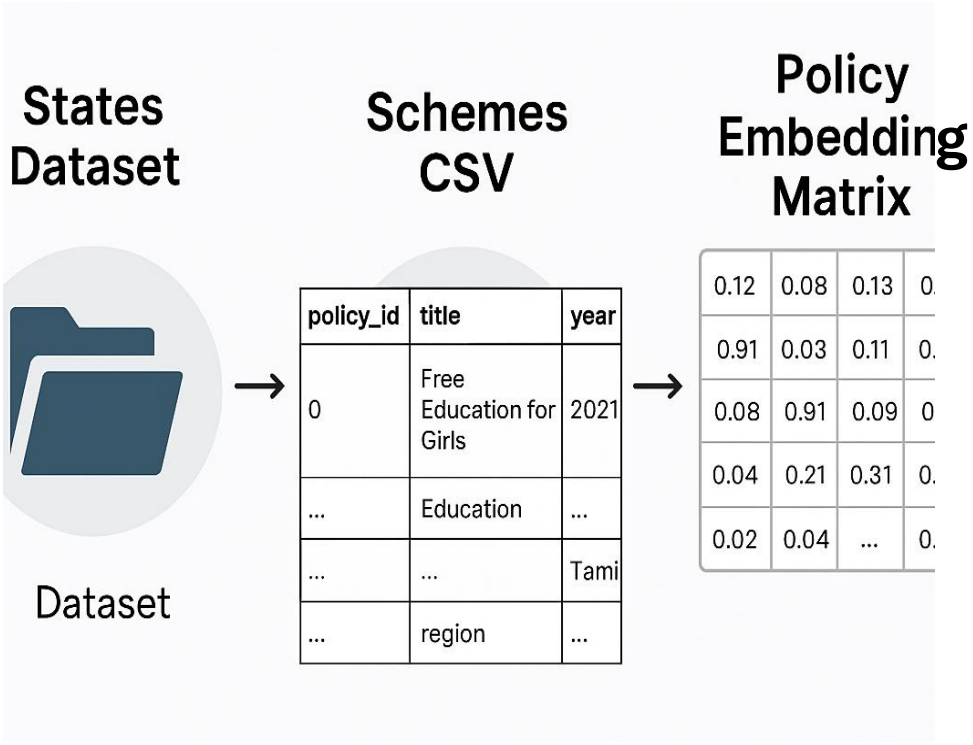
```python
import os
import pandas as pd
import re
import joblib
from sklearn.feature_extraction.text import TfidfVectorizer

DATASET_PATH = "kaggle_dataset"
OUTPUT_CSV = "data/schemes.csv"

def extract_state_from_path(path):
    return os.path.basename(os.path.dirname(path)).replace("-", " ").title()

def extract_year(text):
    match = re.search(r'20\d{2}', text)
    return int(match.group(0)) if match else 2020

def assign_category(text):
    text = text.lower()
    if "education" in text or "student" in text or "fee" in text:
        return "Education"
    elif "women" in text or "girl" in text:
        return "Women Welfare"
    elif "health" in text or "hospital" in text:
        return "Health"
    elif "agriculture" in text or "farmer" in text:
        return "Agriculture"
    elif "transport" in text or "vehicle" in text:
        return "Transport"
    else:
        return "Other"

records = []
for root, dirs, files in os.walk(DATASET_PATH):
    for filename in files:
        if filename.endswith(".txt"):
            filepath = os.path.join(root, filename)
            with open(filepath, "r", encoding="utf-8") as f:
                content = f.read().strip()
                lines = content.split("\n")
                title = lines[0] if lines else "Unknown Title"
                state = extract_state_from_path(filepath)
                year = extract_year(content)
                category = assign_category(content)
                records.append({
                    "policy_id": len(records),
                    "title": title,
                    "full_text": content,
                    "state": state,
                    "year": year,
                    "category": category,
                    "status": "Active",
                    "region": state
                })

df = pd.DataFrame(records)
os.makedirs("data", exist_ok=True)
df.to_csv(OUTPUT_CSV, index=False)
print(f"[INFO] Saved CSV to {OUTPUT_CSV}")

vectorizer = TfidfVectorizer(max_features=5000, stop_words='english')
tfidf_matrix = vectorizer.fit_transform(df['full_text'].values)

os.makedirs("app/models", exist_ok=True)
joblib.dump(vectorizer, "app/models/policy_vectorizer.pkl")
joblib.dump({"matrix": tfidf_matrix, "df": df}, "app/models/policy_tfidf_matrix.pkl")
print("[INFO] TF-IDF vectorizer and matrix saved successfully!")
```

Preprocess and train Data

1.2

OVERVIEW

```python
import os
import pandas as pd
import joblib

def load_policy_data(dataset_path=None, models_path=None):
    BASE_DIR = os.path.dirname(os.path.abspath(__file__))

    # ---------------- Default paths ---------------- #
    if dataset_path is None:
        dataset_path = os.path.join(BASE_DIR, "..", "data", "schemes.csv")
    if models_path is None:
        models_path = os.path.join(BASE_DIR, "models")

    dataset_path = os.path.abspath(dataset_path)
    models_path = os.path.abspath(models_path)

    # ---------------- Load CSV ---------------- #
    if not os.path.exists(dataset_path):
        raise FileNotFoundError(f"CSV file not found: {dataset_path}")

    df = pd.read_csv(dataset_path)

    # Ensure required columns exist
    for col in ["policy_id", "title", "full_text", "state", "year", "category", "status", "region"]:
        if col not in df.columns:
            df[col] = None  # Fill missing columns with None

    # ---------------- Load models ---------------- #
    vectorizer_file = os.path.join(models_path, "policy_vectorizer.pkl")
    tfidf_matrix_file = os.path.join(models_path, "policy_tfidf_matrix.pkl")

    if not os.path.exists(vectorizer_file) or not os.path.exists(tfidf_matrix_file):
        raise FileNotFoundError("Vectorizer or TF-IDF matrix not found in models folder")

    vectorizer = joblib.load(vectorizer_file)
    tfidf_data = joblib.load(tfidf_matrix_file)
    tfidf_matrix = tfidf_data["matrix"]

    return df, vectorizer, tfidf_matrix
```



CREATING FRONTEND & BACKEND

1.3

OVERVIEW

Our model integration workflow ensures seamless transition from development to deployment, delivering real-time policy insights to end users.

1

Train & Store

Model training completed and serialized to model.pkl file

2

Load in API

FastAPI backend loads the pickled model at startup

3

Process Query

User query received, preprocessed, and fed to model

4

Generate Prediction

Model analyzes query and retrieves relevant policy information

5

Return Result

Response formatted and displayed on frontend interface

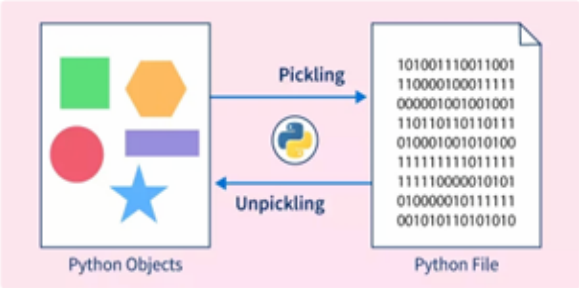
Key Advantage

Real-time policy query resolution using AI eliminates the need for manual document searches, reducing hours of work to seconds of interaction.

Model Persistence with Pickle

What is Pickle?

Pickle is Python's serialization module that saves trained models to disk. This eliminates the need for retraining, reduces computation time, and ensures reproducibility across deployments.



Implementation

```
import pickle

# Save trained model
with open("policy_model.pkl", "wb") as f:
    pickle.dump(model, f)

# Load model for predictions
with open("policy_model.pkl", "rb") as f:
    model = pickle.load(f)

# Now ready for deployment
```

This workflow enables instant model loading at server startup, providing millisecond response times for user queries.

Efficiency

No retraining required for each deployment

Reproducibility

Consistent predictions across environments

Performance

Fast loading and real-time inference

Application Architecture

Project Structure

```
PolicyNav/
│
├── app/
│   ├── main.py
│   ├── routes/
│   ├── models/
│   │   └── model.pkl
│   ├── static/
│   │   ├── css/
│   │   └── js/
│   └── templates/
│       └── index.html
│
└── requirements.txt
```

FastAPI Backend

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def home():
    return {
        "message": "Welcome to PolicyNav"
    }

@app.post("/predict")
def predict(query: str):
    result = model.predict([query])
    return {"response": result}
```

Run with: `uvicorn main:app --reload`

Frontend Role

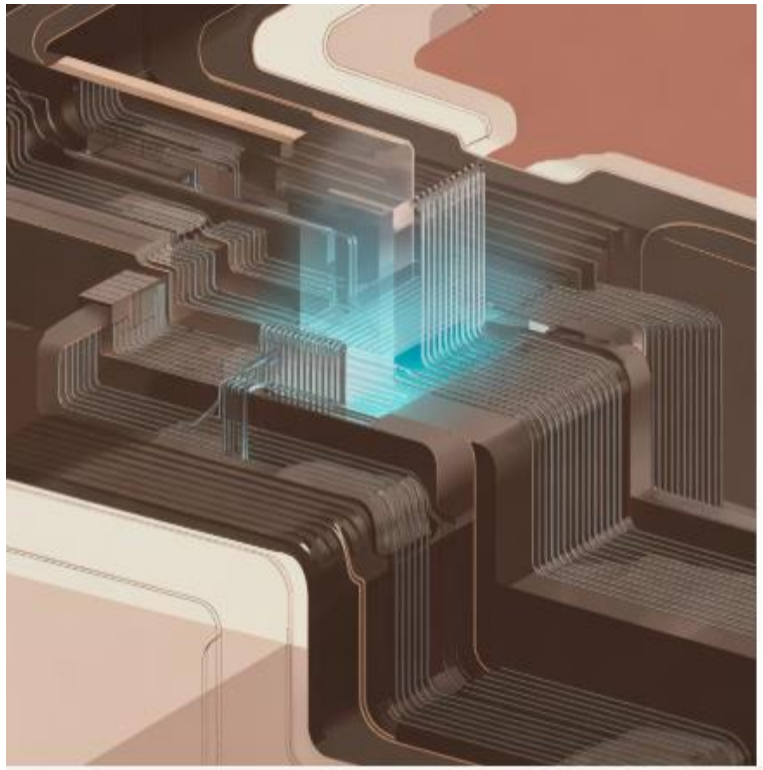
Captures user input, sends API requests, and displays AI-generated responses dynamically

Backend Processing

Handles query preprocessing, model inference, and response formatting

Model Integration

Seamlessly connects trained NLP model with web interface



Quantum NLP for Policy Analysis

Integrating quantum intelligence with classical NLP to revolutionize how we understand and navigate complex policy documents through advanced semantic analysis.

PART 2

Bringing Quantum Intelligence to Policy Analysis

Classical NLP Approach

Traditional embeddings use numeric vectors (TF-IDF, BERT) to capture semantic meaning in policy text.

- Fixed dimensionality
- Linear relationships
- Limited context capture

Quantum NLP Enhancement

Quantum states encode text data with **superposition** and **entanglement**, enabling richer vector relationships.

- Exponential state space
- Non-linear correlations
- Enhanced similarity detection

Powered by **PennyLane**, our hybrid framework seamlessly links Python ML pipelines with quantum circuits to improve contextual understanding of complex policy documents.

Hybrid Quantum-Classical Pipeline

Text Input

Policy documents tokenized into sentence embeddings

Angle Encoding

Word vectors converted into qubit rotations

Quantum Circuit

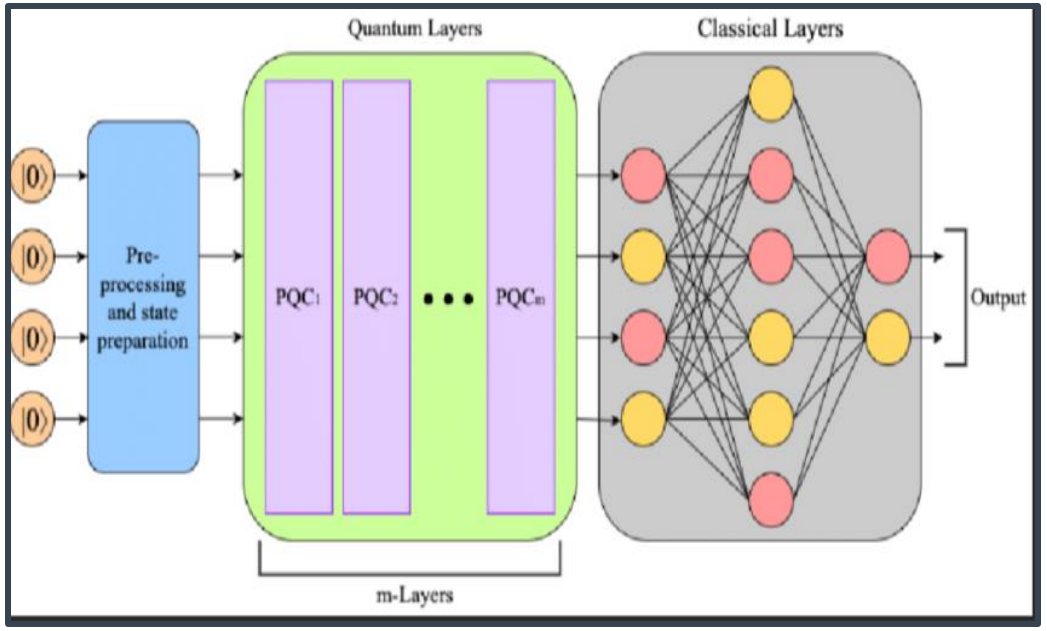
Entanglement layers capture relational learning

Measurement

Quantum output feeds classical model

Prediction

Combined model enhances semantic matching



Quantum Circuit Implementation

PennyLane Code: Quantum Layer

```
import pennylane as qml

dev = qml.device("default.qubit", wires=4)

@qml.qnode(dev)
def quantum_layer(inputs, weights):
    qml.AngleEmbedding(inputs, wires=range(4))
    qml.BasicEntanglerLayers(weights, wires=range(4))
    return qml.expval(qml.PauliZ(0))
```

AngleEmbedding

Maps text embeddings directly to qubit rotation angles, encoding semantic information in quantum states.

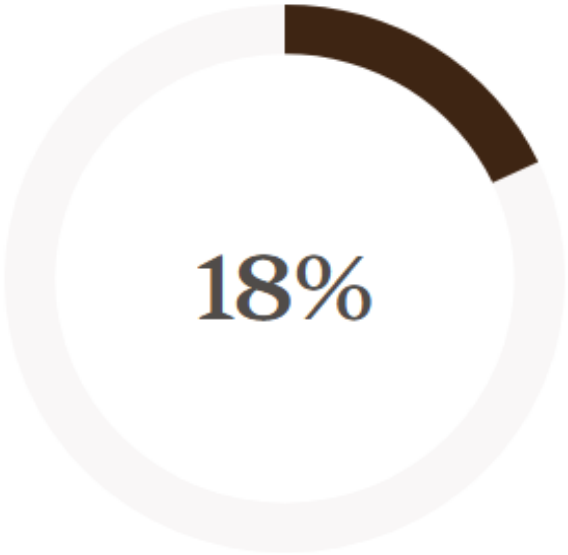
Entangler Layers

Captures cross-term dependencies between words through quantum entanglement operations.

Measurement Output

Produces scalar features via Pauli-Z expectation values for downstream ML integration.

Performance Insights



Accuracy Improvement

Enhanced semantic sensitivity on ambiguous policy queries

Millisecond Latency

Acceptable overhead for real-time policy search applications

Qubits Used

Demonstrates scalability potential for larger circuits

> Quantum embeddings demonstrate early promise of **Quantum Machine Learning** in NLP, particularly for applications requiring nuanced understanding of complex policy language and ambiguous queries.

INTERGRATING AI

PART 3

AI Integration Architecture

Connecting TinyLlama with PolicyNav Backend

Voice/Text Query

User input captured

LM Studio

TinyLlama processing

FastAPI

Request handling

Policy DB

Document retrieval

Response

Natural language output

Key Functions

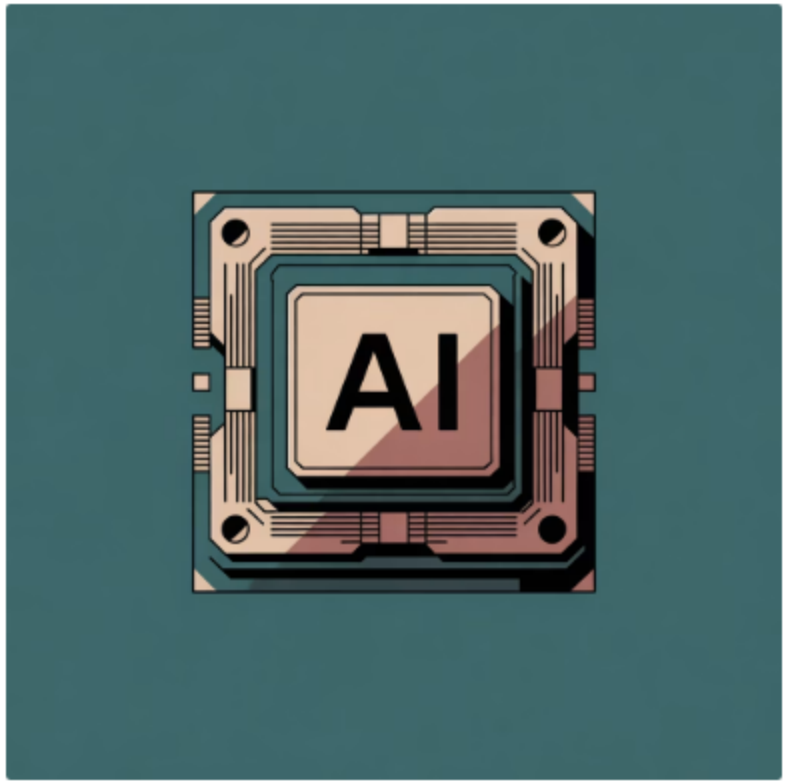
- FastAPI manages API calls and response formatting
- TinyLlama generates policy summaries with contextual understanding
- Supports both typed queries and voice-driven questions

Example Interaction

User: "What reforms were proposed in NEP 2020 for teacher training?"

TinyLlama: "NEP 2020 recommends continuous professional development and technology-integrated teacher education programs."

TinyLlama: Efficient Conversational LLM



Compact Yet Powerful

TinyLlama is a **7-billion parameter** large language model trained on high-quality corpora, optimized for efficient inference without sacrificing conversational capabilities.

On-Device Inference

Runs locally for privacy-safe interactions

Policy Dialogue

Generates natural language summaries

REST Integration

Seamless FastAPI backend connection

VOICE TO TEXT QUERY

PART 4

Voice-Driven Policy Navigation

1

Speech Capture

Web Speech API converts user speech to text in real-time through browser-native functionality

2

Query Transmission

JavaScript automatically sends transcribed text to FastAPI /predict endpoint for processing

3

Model Processing

Backend routes query through classical, quantum, or TinyLlama models based on complexity

4

Voice Response

Browser Text-to-Speech API vocalizes policy answers for seamless hands-free interaction

Implementation Snippet

```
const recognition = new webkitSpeechRecognition();
recognition.onresult = (e) => {
 document.getElementById("query").value =
 e.results[0][0].transcript;
};
recognition.start();
```

OTHER FEATURES

PART 4

Interactive Data Visualization

Policy Trends Dashboard

This dynamic dashboard leverages **Chart.js** to visually represent key policy insights, including most-queried topics, sentiment trends, and year-wise reforms.

Charts update dynamically via **AJAX** calls to FastAPI's `/visualize` endpoint

Enhances data comprehension with flexible bar, pie, and line charts

Tech Stack

- Frontend: `JavaScript` + `Chart.js`
- Backend: `FastAPI` route `/visualize`
- Data Source: Processed `CSV/JSON` files

```
fetch('/visualize')
  .then(res => res.json())
  .then(data => new Chart(ctx, { type: 'bar',
data }));
```

Smart Policy Exporter

Secure Data Export

The **Smart Policy Exporter** allows users to download filtered policy analysis results in both **JSON** and **PDF** formats. This enables quick sharing, offline review, and integration with other systems.

FastAPI's robust response utilities ensure secure and efficient delivery of these files, enhancing accessibility to crucial policy insights.

Tech Stack:

- Backend: `FastAPI /download/json` & `/download/pdf`
- Libraries: `reportlab` for PDF, `json` for JSON

Download Endpoints

```
@app.get("/download/json")
def download_json():
    return
JSONResponse(content=policy_data)

@app.get("/download/pdf")
def download_pdf():
    # generate PDF dynamically
    return
FileResponse("policy_summary.pdf")
```

JSON Export

Structured policy data for programmatic use and data analysis.

PDF Report

Print-ready, human-readable summaries for presentations and reviews.

Easy Sharing

Facilitates collaboration and distribution of findings securely.

Impact & Future Vision

Key Results

- Successfully identifies relevant policy sections with high accuracy
- Reduces manual search effort from hours to seconds
- Improved performance through advanced preprocessing techniques
- Scalable architecture ready for real-world deployment

Future Enhancements

- Integrate transformer-based LLMs (GPT, T5) for advanced QA
- Multi-language support for regional Indian policies
- Voice-based query system for accessibility
- Vector databases (FAISS, Pinecone) for semantic search
- Cloud deployment on Azure/AWS for scalability

Simplifying Policy. Empowering Citizens.

PolicyNav demonstrates how AI and NLP can democratize access to complex policy information, contributing to digital governance and enhanced citizen engagement. Our modular, scalable architecture serves as a foundation for comprehensive policy analysis across multiple domains.