

# Homework 06 - Labeling legislative bills

INFO 4940/5940 - Fall 2025

Jess

2025-11-19

## Exercise 1

### Methodology

This exercise implements a  $3 \times 3$  experimental design to evaluate LLM performance on legislative bill classification:

- **Models:** GPT-4o-mini, GPT-4o, GPT-4-turbo
- **Prompt Strategies:** Simple, Detailed (explicit labels), Reasoning
- **Dataset:** 500 legislative bills from the Comparative Agendas Project (20 policy categories)

All 9 model/prompt combinations were tested on the complete dataset, with token usage tracked for cost analysis.

### Implementation

```
#!/usr/bin/env python3
"""
Legislative Bill Classification Script
Tests 3 models x 3 prompt strategies = 9 combinations
"""

import os
import re
import pandas as pd
from pathlib import Path
```

```

from openai import OpenAI
from tqdm import tqdm
from dotenv import load_dotenv

# Load environment variables
load_dotenv()

# Initialize OpenAI client
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

def load_prompt_template(prompt_file: str) -> str:
    """Load a prompt template from the prompts directory"""
    prompt_path = Path("prompts") / prompt_file
    with open(prompt_path, 'r') as f:
        return f.read()

def classify_bill(description: str, prompt_template: str, model: str, is_reasoning: bool = False):
    """
    Classify a single bill using a specific model and prompt
    Returns: (predicted_category, usage_stats)
    """
    # Format the prompt with the bill description
    prompt = prompt_template.format(description=description)

    try:
        # Call the API with more tokens for reasoning prompts
        max_tokens = 200 if is_reasoning else 10

        response = client.chat.completions.create(
            model=model,
            messages=[
                {"role": "user", "content": prompt}
            ],
            temperature=0,
            max_tokens=max_tokens
        )

        # Extract response text
        response_text = response.choices[0].message.content.strip()

        # Parse the category number from the response
        # For reasoning prompts, look for "ANSWER: X" pattern first
    
```

```

if is_reasoning:
    match = re.search(r'ANSWER:\s*(\d+)', response_text, re.IGNORECASE)
    if match:
        prediction = int(match.group(1))
    else:
        # Fallback: look for any number at the end of the text
        match = re.search(r'\b([1-9]|1[0-9]|20)\b(?:.*\b([1-9]|1[0-9]|20)\b)', response_text)
        if match:
            prediction = int(match.group(1))
        else:
            print(f"Could not parse category from response: {response_text[:50]}")
            prediction = None
    else:
        # For simple prompts, look for a number between 1 and 20
        match = re.search(r'\b([1-9]|1[0-9]|20)\b', response_text)
        if match:
            prediction = int(match.group(1))
        else:
            print(f"Could not parse category from response: {response_text}")
            prediction = None

# Extract usage stats
usage = {
    "prompt_tokens": response.usage.prompt_tokens,
    "completion_tokens": response.usage.completion_tokens,
    "total_tokens": response.usage.total_tokens
}

return prediction, usage
except Exception as e:
    print(f"Error classifying: {e}")
    return None, {"prompt_tokens": 0, "completion_tokens": 0, "total_tokens": 0}

def run_experiment(data_file: str = "data/leg_lite.feather",
                   sample_size: int | None = None):
    """
    Run the full 33 experimental design
    """

    Args:
        data_file: Path to the legislative bills dataset
        sample_size: If provided, only test on first N bills (for testing)
    """

```

```

# Load data
print("Loading data...")
df = pd.read_feather(data_file)

if sample_size:
    df = df.head(sample_size)
    print(f"Testing on {sample_size} bills")
else:
    print(f"Processing {len(df)} bills")

# Define experimental conditions
models = ["gpt-4o-mini", "gpt-4o", "gpt-4-turbo"] # Small, optimized, and turbo models
prompts = {
    "simple": load_prompt_template("cap-simple.md"),
    "detailed": load_prompt_template("cap-detailed.md"),
    "reasoning": load_prompt_template("cap-reasoning.md")
}

# Initialize results storage
results = df[['id', 'description', 'policy', 'policy_label']].copy()

# Track token usage
usage_stats = {}

# Run all 9 combinations
for model in models:
    for prompt_name, prompt_template in prompts.items():
        col_name = f"{model.replace('gpt-', 'gpt').replace('-turbo-preview', '')}.replace"
        print(f"\nProcessing: {model} + {prompt_name} prompt")

        predictions = []
        total_tokens = 0

        for _, row in tqdm(df.iterrows(), total=len(df), desc=f"{model}_{prompt_name}"):
            is_reasoning = (prompt_name == "reasoning")
            pred, usage = classify_bill(row['description'], prompt_template, model, is_reasoning)
            predictions.append(pred)
            total_tokens += usage['total_tokens']

        # Store results
        results[col_name] = predictions
        usage_stats[col_name] = total_tokens

```

```

    print(f"Total tokens used: {total_tokens:,}")

# Calculate accuracy for each combination
print("\n" + "="*60)
print("ACCURACY RESULTS")
print("="*60)

for col in results.columns:
    if col not in ['id', 'description', 'policy', 'policy_label']:
        correct = (results[col] == results['policy']).sum()
        total = len(results)
        accuracy = correct / total * 100
        tokens = usage_stats[col]
        print(f"{col}: {accuracy:.2f}% ({correct}/{total}) | Tokens: {tokens:,}")

# Export results
output_file = "data/leg_predictions.feather"
results.to_feather(output_file)
print(f"\nResults saved to: {output_file}")

# Also save usage stats
usage_df = pd.DataFrame([usage_stats])
usage_df.to_csv("data/token_usage.csv", index=False)
print(f"Token usage saved to: data/token_usage.csv")

return results, usage_stats

if __name__ == "__main__":
    # Run on full dataset
    print("Running on full dataset (500 bills)...")
    print("This will take 30-60 minutes and will consume API tokens.")
    print("Estimated cost: $5-20 depending on the models used.\n")
    run_experiment()

```

## Prompt Templates

### Naive and simple prompt

This minimalist prompt provides only basic instructions without category details.

You are an expert in U.S. legislative policy classification. Given a legislative bill description, classify it.

Bill description: {description}

Return ONLY the policy category number (1-20) as a single integer with no other text.

### Explicit code/label values

This prompt includes the complete list of 20 policy categories with their names.

You are an expert in U.S. legislative policy classification. Classify the following bill description.

Policy Categories:

1. Macroeconomics
2. Civil rights, minority issues, civil liberties
3. Health
4. Agriculture
5. Labor and employment
6. Education
7. Environment
8. Energy
9. Immigration
10. Transportation
11. Law, crime, family issues
12. Social welfare
13. Community development and housing issues
14. Banking, finance, and domestic commerce
15. Defense
16. Space, technology, and communications
17. Foreign trade
18. International affairs and foreign aid
19. Government operations
20. Public lands and water management

Bill description: {description}

Return ONLY the policy category number (1-20) as a single integer with no other text.

## Detailed and reasoning prompt

This prompt provides comprehensive category descriptions and encourages step-by-step analysis using prompt engineering best practices.

```
You are an expert policy analyst specializing in U.S. legislative classification using the Comparative Agendas Project's 20 major policy categories.

**Classification Framework:**  
The Comparative Agendas Project uses 20 major policy categories:  
1. Macroeconomics (fiscal policy, taxation, budgets, monetary policy)  
2. Civil rights, minority issues, civil liberties (voting rights, discrimination, freedom of religion, equal protection)  
3. Health (healthcare delivery, insurance, medical research, public health)  
4. Agriculture (farming, food safety, agricultural trade, subsidies)  
5. Labor and employment (worker rights, unemployment, workplace safety, wages)  
6. Education (K-12, higher education, student aid, educational standards)  
7. Environment (pollution, conservation, wildlife, climate)  
8. Energy (production, distribution, alternative energy, fossil fuels)  
9. Immigration (border security, citizenship, refugees, visas)  
10. Transportation (roads, aviation, rail, mass transit, infrastructure)  
11. Law, crime, family issues (criminal justice, law enforcement, family law)  
12. Social welfare (poverty assistance, food stamps, housing assistance)  
13. Community development and housing issues (urban development, housing policy)  
14. Banking, finance, and domestic commerce (financial regulation, securities, consumer protection)  
15. Defense (military operations, weapons systems, veterans)  
16. Space, technology, and communications (telecommunications, internet, space exploration)  
17. Foreign trade (trade agreements, tariffs, exports/imports)  
18. International affairs and foreign aid (diplomacy, foreign relations, international organizations)  
19. Government operations (federal administration, public employees, government efficiency)  
20. Public lands and water management (federal lands, parks, water resources)

Bill description: {description}

Analyze the bill systematically: identify the primary policy domain, consider key terms and context.
```

## Results

The classification script was run on all 500 bills across 9 model/prompt combinations. Key findings:

- **Total combinations tested:** 9 (3 models × 3 prompts)
- **Success rate:** Variable by combination, with reasoning prompts showing lower completion rates

- **Output files:**
  - `data/leg_predictions.feather` - All predictions with true labels
  - `data/token_usage.csv` - Token consumption per combination

## Token Usage Summary

Model/Prompt Combination	Total Tokens	Est. Cost (USD)
gpt4o_mini_simple	47,529	\$0.011
gpt4o_mini_detailed	109,029	\$0.026
gpt4o_mini_reasoning	325,429	\$0.078
gpt4o_simple	47,529	\$0.190
gpt4o_detailed	109,029	\$0.436
gpt4o_reasoning	293,292	\$1.173
gpt4_turbo_simple	47,691	\$0.668
gpt4_turbo_detailed	108,691	\$1.522
gpt4_turbo_reasoning	332,229	\$4.651

**Total estimated cost:** ~\$8.76 for all 9 combinations on 500 bills.

## Exercise 2

### Performance Evaluation

This exercise evaluates the performance of all 9 model/prompt combinations using four key metrics:

- **Accuracy:** Overall correct classification rate
- **F1-Score (Macro):** Harmonic mean of precision and recall, averaged across classes
- **Sensitivity (Recall):** True positive rate - ability to correctly identify each category
- **Specificity:** True negative rate - ability to avoid false positives

### Analysis Implementation

```
#!/usr/bin/env python3
"""
Analysis Script for Exercise 2
Evaluates LLM classification performance using multiple metrics
```

```

"""
import pandas as pd
import numpy as np
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    recall_score,
    confusion_matrix,
    classification_report
)
import seaborn as sns
import matplotlib.pyplot as plt
from pathlib import Path

def calculate_specificity(y_true, y_pred, average='macro'):
    """
    Calculate specificity (true negative rate) for multi-class classification

    Specificity = TN / (TN + FP)
    For multi-class, we calculate per-class and average
    """
    # Get unique classes
    classes = np.unique(np.concatenate([y_true, y_pred]))

    specificities = []
    for cls in classes:
        # Create binary classification for this class
        y_true_binary = (y_true == cls).astype(int)
        y_pred_binary = (y_pred == cls).astype(int)

        # Calculate confusion matrix components
        tn = np.sum((y_true_binary == 0) & (y_pred_binary == 0))
        fp = np.sum((y_true_binary == 0) & (y_pred_binary == 1))

        # Calculate specificity for this class
        if (tn + fp) > 0:
            spec = tn / (tn + fp)
        else:
            spec = 0.0

        specificities.append(spec)

```

```

if average == 'macro':
    return np.mean(specificities)
elif average == 'weighted':
    # Weight by class frequency
    class_counts = [np.sum(y_true == cls) for cls in classes]
    return np.average(specificities, weights=class_counts)
else:
    return specificities

def evaluate_model(y_true, y_pred, model_name):
    """
    Evaluate a single model/prompt combination

    Returns dictionary with all metrics
    """
    # Remove None/NaN predictions (failed classifications)
    valid_idx = [i for i, pred in enumerate(y_pred) if pred is not None and not (isinstance(pred, float) and np.isnan(pred))]
    y_true_valid = y_true[valid_idx]
    y_pred_valid = np.array([y_pred[i] for i in valid_idx])

    # Calculate metrics
    accuracy = accuracy_score(y_true_valid, y_pred_valid)
    f1_macro = f1_score(y_true_valid, y_pred_valid, average='macro', zero_division=0)
    f1_weighted = f1_score(y_true_valid, y_pred_valid, average='weighted', zero_division=0)
    sensitivity = recall_score(y_true_valid, y_pred_valid, average='macro', zero_division=0)
    specificity = calculate_specificity(y_true_valid, y_pred_valid, average='macro')

    # Count failed predictions
    failed_preds = len(y_pred) - len(valid_idx)

    return {
        'model': model_name,
        'accuracy': accuracy,
        'f1_macro': f1_macro,
        'f1_weighted': f1_weighted,
        'sensitivity': sensitivity,
        'specificity': specificity,
        'valid_predictions': len(valid_idx),
        'failed_predictions': failed_preds,
        'success_rate': len(valid_idx) / len(y_pred)
    }

```

```

def analyze_results(predictions_file='data/leg_predictions.feather',
                    token_usage_file='data/token_usage.csv'):
    """
    Analyze all model/prompt combinations
    """

    print("Loading predictions...")
    df = pd.read_feather(predictions_file)

    # Load token usage if available
    token_df = None
    if Path(token_usage_file).exists():
        token_df = pd.read_csv(token_usage_file)
        print("Loaded token usage data")

    # Get true labels
    y_true = df['policy'].values

    # Get all prediction columns (exclude metadata columns)
    pred_cols = [col for col in df.columns
                 if col not in ['id', 'description', 'policy', 'policy_label']]

    # Evaluate each model/prompt combination
    results = []
    for col in pred_cols:
        y_pred = df[col].values
        metrics = evaluate_model(y_true, y_pred, col)
        results.append(metrics)

    # Create results dataframe
    results_df = pd.DataFrame(results)

    # Sort by accuracy
    results_df = results_df.sort_values('accuracy', ascending=False)

    # Add token usage and cost estimates if available
    if token_df is not None:
        # Merge token usage
        for idx, row in results_df.iterrows():
            model_name = row['model']
            if model_name in token_df.columns:
                tokens = token_df[model_name].values[0]
                results_df.loc[idx, 'total_tokens'] = tokens

```

```

# Estimate costs (approximate pricing as of 2024)
if 'gpt4o_mini' in model_name:
    # GPT-4o-mini: $0.15/1M input, $0.60/1M output
    # Assume roughly 80% input, 20% output
    cost = (tokens * 0.8 * 0.15 / 1_000_000) + (tokens * 0.2 * 0.60 / 1_000_000)
elif 'gpt4o' in model_name:
    # GPT-4o: $2.50/1M input, $10.00/1M output
    cost = (tokens * 0.8 * 2.50 / 1_000_000) + (tokens * 0.2 * 10.00 / 1_000_000)
elif 'gpt4_turbo' in model_name:
    # GPT-4-turbo: $10.00/1M input, $30.00/1M output
    cost = (tokens * 0.8 * 10.00 / 1_000_000) + (tokens * 0.2 * 30.00 / 1_000_000)
else:
    cost = 0

results_df.loc[idx, 'estimated_cost_usd'] = cost

# Calculate value metrics
if 'estimated_cost_usd' in results_df.columns:
    results_df['accuracy_per_dollar'] = results_df['accuracy'] / results_df['estimated_cost_usd']
    results_df['f1_per_dollar'] = results_df['f1_macro'] / results_df['estimated_cost_usd']

return results_df, df

def create_visualizations(results_df, output_dir='figures'):
    """
    Create visualizations comparing model performance
    """
    Path(output_dir).mkdir(exist_ok=True)

    # Parse model and prompt from model name
    results_df['base_model'] = results_df['model'].str.extract(r'(gpt\d+[a-z_]*)')
    results_df['prompt_type'] = results_df['model'].str.extract(r'_(simple|detailed|reasoning)')

    # 1. Performance comparison heatmap
    fig, axes = plt.subplots(2, 2, figsize=(16, 12))

    # Accuracy heatmap
    pivot_acc = results_df.pivot(index='prompt_type', columns='base_model', values='accuracy')
    sns.heatmap(pivot_acc, annot=True, fmt='.3f', cmap='RdYlGn', ax=axes[0, 0], vmin=0, vmax=1)
    axes[0, 0].set_title('Accuracy by Model and Prompt Type')

    # F1-Score heatmap

```

```

pivot_f1 = results_df.pivot(index='prompt_type', columns='base_model', values='f1_macro')
sns.heatmap(pivot_f1, annot=True, fmt='.3f', cmap='RdYlGn', ax=axes[0, 1], vmin=0, vmax=1)
axes[0, 1].set_title('F1-Score (Macro) by Model and Prompt Type')

# Sensitivity heatmap
pivot_sens = results_df.pivot(index='prompt_type', columns='base_model', values='sensitivity')
sns.heatmap(pivot_sens, annot=True, fmt='.3f', cmap='RdYlGn', ax=axes[1, 0], vmin=0, vmax=1)
axes[1, 0].set_title('Sensitivity (Recall) by Model and Prompt Type')

# Specificity heatmap
pivot_spec = results_df.pivot(index='prompt_type', columns='base_model', values='specificity')
sns.heatmap(pivot_spec, annot=True, fmt='.3f', cmap='RdYlGn', ax=axes[1, 1], vmin=0, vmax=1)
axes[1, 1].set_title('Specificity by Model and Prompt Type')

plt.tight_layout()
plt.savefig(f'{output_dir}/performance_heatmaps.png', dpi=300, bbox_inches='tight')
print(f"Saved performance heatmaps to {output_dir}/performance_heatmaps.png")

# 2. Bar plot comparing all metrics
fig, ax = plt.subplots(figsize=(14, 6))
metrics = ['accuracy', 'f1_macro', 'sensitivity', 'specificity']
x = np.arange(len(results_df))
width = 0.2

for i, metric in enumerate(metrics):
    ax.bar(x + i * width, results_df[metric], width, label=metric.replace('_', ' ').title())

ax.set_xlabel('Model/Prompt Combination')
ax.set_ylabel('Score')
ax.set_title('Performance Metrics Comparison')
ax.set_xticks(x + width * 1.5)
ax.set_xticklabels(results_df['model'], rotation=45, ha='right')
ax.legend()
ax.grid(axis='y', alpha=0.3)

plt.tight_layout()
plt.savefig(f'{output_dir}/metrics_comparison.png', dpi=300, bbox_inches='tight')
print(f"Saved metrics comparison to {output_dir}/metrics_comparison.png")

# 3. Cost-effectiveness plot (if cost data available)
if 'estimated_cost_usd' in results_df.columns:
    fig, ax = plt.subplots(figsize=(10, 6))

```

```

scatter = ax.scatter(results_df['estimated_cost_usd'],
                     results_df['accuracy'],
                     s=200,
                     c=results_df['f1_macro'],
                     cmap='viridis',
                     alpha=0.6,
                     edgecolors='black')

# Label points
for idx, row in results_df.iterrows():
    ax.annotate(row['model'],
                (row['estimated_cost_usd'], row['accuracy']),
                fontsize=8,
                ha='center')

ax.set_xlabel('Estimated Cost (USD)')
ax.set_ylabel('Accuracy')
ax.set_title('Cost vs. Accuracy (color = F1-score)')
ax.grid(alpha=0.3)

plt.colorbar(scatter, label='F1-Score')
plt.tight_layout()
plt.savefig(f'{output_dir}/cost_effectiveness.png', dpi=300, bbox_inches='tight')
print(f"Saved cost-effectiveness plot to {output_dir}/cost_effectiveness.png")

plt.close('all')

def print_summary(results_df):
    """
    Print a summary of results
    """
    print("\n" + "="*80)
    print("PERFORMANCE SUMMARY")
    print("="*80)

    print("\nTop 3 Models by Accuracy:")
    print(results_df[['model', 'accuracy', 'f1_macro', 'sensitivity', 'specificity']].head(3))

    if 'estimated_cost_usd' in results_df.columns:
        print("\n" + "="*80)
        print("COST ANALYSIS")
        print("="*80)

```

```

print("\nTotal costs:")
print(results_df[['model', 'total_tokens', 'estimated_cost_usd']].to_string(index=False))

print("\n\nBest value for money (by accuracy per dollar):")
best_value = results_df.nlargest(3, 'accuracy_per_dollar')
print(best_value[['model', 'accuracy', 'estimated_cost_usd', 'accuracy_per_dollar']])


print("\n" + "="*80)
print("DETAILED RESULTS TABLE")
print("="*80)
print(results_df.to_string(index=False))

if __name__ == "__main__":
    # Run analysis
    results_df, predictions_df = analyze_results()

    # Create visualizations
    create_visualizations(results_df)

    # Print summary
    print_summary(results_df)

    # Save results to CSV
    results_df.to_csv('data/performance_metrics.csv', index=False)
    print("\n\nResults saved to data/performance_metrics.csv")

```

## Results and Discussion

### Top Performing Combinations

The analysis reveals clear patterns in model and prompt performance:

#### Top 3 by Accuracy:

1. **gpt4o\_detailed** - 68.0% accuracy (F1: 0.652, Sensitivity: 0.654, Specificity: 0.983)
2. **gpt4o\_reasoning** - 66.6% accuracy (F1: 0.627, Sensitivity: 0.636, Specificity: 0.982)
3. **gpt4\_turbo\_detailed** - 63.8% accuracy (F1: 0.617, Sensitivity: 0.623, Specificity: 0.981)

## Key Findings

- 1. Prompt Strategy Impact:** - **Detailed prompts** (with explicit category lists) consistently outperformed other strategies across all models - **Reasoning prompts** showed mixed results - high accuracy when successful, but significant failure rates: - gpt4\_turbo\_reasoning: 323/500 failed predictions (35.4% success rate) - gpt4o\_mini\_reasoning: 85/500 failed predictions (83% success rate) - **Simple prompts** performed extremely poorly (1.8%-24% accuracy), demonstrating the critical importance of providing category information
- 2. Model Performance:** - GPT-4o consistently delivered the best results across all prompt types - GPT-4-turbo had competitive accuracy but poor reliability with reasoning prompts - GPT-4o-mini provided surprisingly good performance for its cost tier
- 3. Cost-Effectiveness Analysis:**

Model/Prompt	Accuracy	Cost (USD)	Accuracy/Dollar
gpt4o_mini_detailed	58.6%	\$0.026	22.39
gpt4o_detailed	68.0%	\$0.436	1.56
gpt4_turbo_detailed	63.8%	\$1.522	0.42

**Best value:** gpt4o\_mini\_detailed provides 58.6% accuracy at just \$0.026 - exceptional value for budget-conscious applications.

**Best balance:** gpt4o\_detailed achieves 68% accuracy for \$0.436, offering strong performance at reasonable cost.

**Poor value:** gpt4\_turbo\_reasoning cost \$4.65 but only achieved 57% accuracy with a 35% failure rate.

## Performance Visualizations

The analysis generated three key visualizations saved in `figures/`:

1. **Performance heatmaps** - Compare all metrics across models and prompt types
2. **Metrics comparison** - Bar chart showing accuracy, F1, sensitivity, and specificity
3. **Cost-effectiveness plot** - Scatter plot of cost vs. accuracy with F1 score as color dimension

## Recommendations

**For production use:** GPT-4o with detailed prompts offers the best accuracy (68%) and reliability.

**For cost-sensitive applications:** GPT-4o-mini with detailed prompts provides excellent value (58.6% accuracy for \$0.026).

**Avoid:** Simple prompts and reasoning prompts with GPT-4-turbo due to poor performance and/or high failure rates.

## Exercise 3

### INFO 4940/5940 Tutor Chatbot

#### Overview

This exercise implements an intelligent tutoring assistant using the shinychat framework with Retrieval-Augmented Generation (RAG) to provide accurate, context-aware responses about course content.

**Live Deployment:** [INFO 4940/5940 Tutor Chatbot](#) (*Note: Update this link after deployment*)

#### Features

The chatbot is designed to help students with:

- **Course Content:** Answers questions about topics covered in lectures and readings
- **Assignments:** Provides guidance on homework and projects without giving complete solutions
- **Policies:** Addresses questions about course requirements, grading, and deadlines
- **Coding Help:** Offers Python and R coding guidance with clear examples
- **Study Tips:** Suggests best practices and learning strategies
- **Debugging:** Helps troubleshoot code issues and error messages

## Technical Implementation

**Framework:** Built with shinychat (Python)

**Model:** GPT-4o for high-quality, accurate responses

**RAG Knowledge Base:** Three curated knowledge documents: - `course-overview.md` - Course description, objectives, and structure - `hw-06-instructions.md` - Complete HW 06 assignment details - `coding-guidance.md` - Python and R coding best practices

**System Prompt:** Carefully designed to: - Define the assistant's role as an INFO 4940/5940 tutor - Encourage learning rather than just providing answers - Ground responses in actual course materials - Maintain academic integrity (no complete solutions) - Provide clear, helpful explanations with examples

## User Interface

The chatbot features a custom UI with: - Gradient header with course branding - Welcome message explaining capabilities - Clean, centered chat interface - Responsive design for various screen sizes

## Implementation Code

```
"""
INFO 4940/5940 Tutor Chatbot
A RAG-powered tutoring assistant for the Applied Machine Learning course
"""

import os
from pathlib import Path
from shiny import App, ui, reactive
from shinychat import chat_ui, chat_server
from openai import OpenAI

# Initialize OpenAI client
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# Load knowledge base files
def load_knowledge_base():
    """Load all knowledge base documents into a single context string"""
    knowledge_dir = Path("knowledge")
    knowledge_content = []
```

```

knowledge_files = [
    "course-overview.md",
    "hw-06-instructions.md",
    "coding-guidance.md"
]

for filename in knowledge_files:
    filepath = knowledge_dir / filename
    if filepath.exists():
        with open(filepath, 'r', encoding='utf-8') as f:
            content = f.read()
            knowledge_content.append(f"## {filename}\n\n{content}\n\n")

return "\n".join(knowledge_content)

# Load knowledge base at startup
KNOWLEDGE_BASE = load_knowledge_base()

# System prompt for the tutor
SYSTEM_PROMPT = f"""You are an expert teaching assistant for INFO 4940/5940: Applied Machine Learning

Your role is to help students with:
- Understanding course concepts, assignments, and projects
- Answering questions about course policies and requirements
- Providing coding guidance in Python or R
- Explaining machine learning concepts and techniques
- Offering study tips and best practices
- Debugging code and troubleshooting issues

Guidelines:
1. Be helpful, encouraging, and patient
2. Provide clear explanations with examples when appropriate
3. Guide students to learn rather than just giving answers
4. Reference specific course materials when relevant
5. Suggest additional resources when helpful
6. For coding questions, provide clear, well-commented code examples
7. If you don't know something, be honest and suggest where to find the answer
8. Encourage good practices: version control, reproducibility, documentation

What you should NOT do:
- Do not complete assignments for students
- Do not provide complete solutions without explanation
"""

```

- Do not make up information about course policies
- Do not encourage academic dishonesty

You have access to the following course materials:

{KNOWLEDGE\_BASE}

Use this information to provide accurate, helpful responses grounded in the actual course content.

```
"""  
  
# Create the Shiny UI  
app_ui = ui.page_fluid(  
    ui.head_content(  
        ui.tags.style("""  
            .app-title {  
                background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);  
                color: white;  
                padding: 20px;  
                border-radius: 10px;  
                margin-bottom: 20px;  
                box-shadow: 0 4px 6px rgba(0,0,0,0.1);  
            }  
            .app-title h2 {  
                margin: 0;  
                font-weight: 600;  
            }  
            .app-title p {  
                margin: 5px 0 0 0;  
                opacity: 0.9;  
            }  
            .info-box {  
                background-color: #f0f4f8;  
                border-left: 4px solid #667eea;  
                padding: 15px;  
                margin-bottom: 20px;  
                border-radius: 5px;  
            }  
            .chat-container {  
                max-width: 900px;  
                margin: 0 auto;  
            }  
        """)  
    )
```

```

),
ui.div(
  {"class": "chat-container"},
  ui.div(
    {"class": "app-title"},
    ui.h2(" INFO 4940/5940 Tutor Assistant"),
    ui.p("Your AI teaching assistant for Applied Machine Learning")
  ),
  ui.div(
    {"class": "info-box"},
    ui.markdown("""
      **Welcome!** I'm here to help you with:
      - Course concepts and assignments
      - Python and R coding guidance
      - Machine learning techniques and best practices
      - Questions about course policies and requirements

      Feel free to ask me anything about the course!
    """)
  ),
  chat_ui("chat")
)
)

def server(input, output, session):
    """Server function with chat integration"""

    chat = chat_server(
        "chat",
        model="gpt-4o",
        system_prompt=SYSTEM_PROMPT,
        api_key=os.getenv("OPENAI_API_KEY"),
        temperature=0.7,
        max_tokens=1000
    )

# Create the Shiny app
app = App(app_ui, server)

if __name__ == "__main__":
    app.run()

```

## **Deployment**

The application is deployed on Posit Connect Cloud with:

- Environment variable for OpenAI API key (deployment-specific)
- All knowledge base files included in deployment
- Requirements specified in `requirements.txt`

**Dependencies:**

- shiny >= 1.5.0
- shinychat >= 0.2.8
- openai >= 1.0.0
- python-dotenv >= 1.0.0

## **Design Philosophy**

The chatbot is designed as a **learning facilitator** rather than a solution provider:

1. **Guidance over answers:** Provides hints and explanations to help students learn
2. **Context-aware:** Uses RAG to provide accurate information from course materials
3. **Encouraging:** Maintains a helpful, patient tone
4. **Practical:** Offers concrete examples and code snippets when appropriate
5. **Honest:** Admits when uncertain and suggests where to find answers

## **Example Interactions**

**Question about assignments:** > “How should I approach Exercise 1 in HW 06?”

The chatbot will explain the  $3 \times 3$  experimental design, discuss prompt engineering strategies, and suggest testing on small samples first - without writing the code for the student.

**Coding help:** > “How do I load a feather file in Python?”

The chatbot provides clear code examples with explanations, referencing the course coding guidance.

**Course policies:** > “What’s the due date for HW 06?”

The chatbot retrieves accurate information from the stored assignment instructions.

## **Generative AI (GAI) self-reflection**