**Requirements file**

torch==2.1.0

transformers==4.36.0

numpy==1.26.4

pandas==2.2.0

scikit-learn==1.4.0

nltk==3.8.1

tqdm==4.66.1

python-dotenv==1.0.0

streamlit==1.29.0

datasets==2.15.0

nest-asyncio==1.6.0

evaluate==0.4.1

rouge-score==0.1.2

accelerate==0.26.0

protobuf==3.20.0

chardet>=5.0.0

docx2txt>=0.8

PyPDF2>=3.0.0

networkx>=3.0

**Utilities file**

*from* typing *import* List, Dict, Union

*import* nltk

*from* nltk.tokenize *import* word\_tokenize, sent\_tokenize

*from* nltk.corpus *import* stopwords

*import* re

*import* logging

*# Download required NLTK data*

*try*:

    nltk.download('punkt')

    nltk.download('stopwords')

*except* Exception *as* e:

    logging.error(f"Error downloading NLTK data: {str(e)}")

def clean\_text(*text*: str) -> str:

    """

    Clean and preprocess the input text.

    Args:

        text (str): Input text

    Returns:

        str: Cleaned text

    """

*# Convert to lowercase*

*text* = *text*.lower()

*# Remove special characters and digits*

*text* = re.sub(r'[^\w\s]', '', *text*)

*text* = re.sub(r'\d+', '', *text*)

*# Remove extra whitespace*

*text* = ' '.join(*text*.split())

*return* *text*

def extract\_key\_sentences(*text*: str, *num\_sentences*: int = 5) -> List[str]:

    """

    Extract key sentences from the text based on importance.

    Args:

        text (str): Input text

        num\_sentences (int): Number of key sentences to extract

    Returns:

        List[str]: List of key sentences

    """

*# Tokenize into sentences*

    sentences = sent\_tokenize(*text*)

*# Calculate sentence scores based on word frequency*

    word\_freq = {}

    stop\_words = set(stopwords.words('english'))

*for* sentence *in* sentences:

        words = word\_tokenize(sentence.lower())

*for* word *in* words:

*if* word not in stop\_words and word.isalnum():

                word\_freq[word] = word\_freq.get(word, 0) + 1

*# Score sentences*

    sentence\_scores = {}

*for* sentence *in* sentences:

        words = word\_tokenize(sentence.lower())

        score = sum(word\_freq.get(word, 0) *for* word *in* words *if* word *not* *in* stop\_words)

        sentence\_scores[sentence] = score

*# Get top sentences*

    key\_sentences = sorted(sentence\_scores.items(), *key*=lambda *x*: *x*[1], *reverse*=True)

*return* [sent *for* sent, score *in* key\_sentences[:*num\_sentences*]]

def calculate\_rouge\_scores(*reference*: str, *candidate*: str) -> Dict[str, float]:

    """

    Calculate ROUGE scores between reference and candidate summaries.

    Args:

        reference (str): Reference summary

        candidate (str): Generated summary

    Returns:

        Dict[str, float]: Dictionary containing ROUGE scores

    """

*# Implement ROUGE score calculation*

*# This is a simplified version - you might want to use a proper ROUGE implementation*

    ref\_tokens = set(word\_tokenize(*reference*.lower()))

    cand\_tokens = set(word\_tokenize(*candidate*.lower()))

*# Calculate F1 score*

    common\_tokens = ref\_tokens.intersection(cand\_tokens)

    precision = len(common\_tokens) / len(cand\_tokens) *if* cand\_tokens *else* 0

    recall = len(common\_tokens) / len(ref\_tokens) *if* ref\_tokens *else* 0

    f1 = 2 \* (precision \* recall) / (precision + recall) *if* (precision + recall) > 0 *else* 0

*return* {

        "rouge\_1\_f1": f1,

        "rouge\_1\_precision": precision,

        "rouge\_1\_recall": recall

    }

def format\_legal\_citations(*text*: str) -> str:

    """

    Format legal citations in the text according to standard conventions.

    Args:

        text (str): Input text containing legal citations

    Returns:

        str: Text with properly formatted citations

    """

*# Add citation formatting logic here*

*# This is a placeholder - implement actual citation formatting rules*

*return* *text*

def extract\_legal\_entities(*text*: str) -> Dict[str, List[str]]:

    """

    Extract legal entities (e.g., cases, statutes, parties) from the text.

    Args:

        text (str): Input text

    Returns:

        Dict[str, List[str]]: Dictionary containing different types of legal entities

    """

    entities = {

        "cases": [],

        "statutes": [],

        "parties": [],

        "dates": []

    }

*# Add entity extraction logic here*

*# This is a placeholder - implement actual entity extraction rules*

*return* entities

def calculate\_readability\_metrics(*text*: str) -> Dict[str, float]:

    """

    Calculate readability metrics for the text.

    Args:

        text (str): Input text

    Returns:

        Dict[str, float]: Dictionary containing readability scores

    """

    sentences = sent\_tokenize(*text*)

    words = word\_tokenize(*text*)

*# Calculate basic metrics*

    num\_sentences = len(sentences)

    num\_words = len(words)

    num\_chars = len(*text*)

*# Average sentence length*

    avg\_sentence\_length = num\_words / num\_sentences *if* num\_sentences > 0 *else* 0

*# Average word length*

    avg\_word\_length = num\_chars / num\_words *if* num\_words > 0 *else* 0

*return* {

        "avg\_sentence\_length": avg\_sentence\_length,

        "avg\_word\_length": avg\_word\_length,

        "num\_sentences": num\_sentences,

        "num\_words": num\_words

    }

**Train File**

*from* transformers *import* AutoModelForSeq2SeqLM, AutoTokenizer, Seq2SeqTrainingArguments, Seq2SeqTrainer, DataCollatorForSeq2Seq, IntervalStrategy

*from* datasets *import* load\_dataset

*import* torch

*import* numpy *as* np

*from* typing *import* Dict, List

*import* evaluate

*import* nltk

*import* logging

*from* pathlib *import* Path

*# Configure logging*

logging.basicConfig(*level*=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

class LegalDocumentTrainer:

    def **\_\_init\_\_**(

*self*,

*model\_name*: str = "facebook/bart-base",

*output\_dir*: str = "fine\_tuned\_model",

*max\_input\_length*: int = 512,

*max\_target\_length*: int = 128

    ):

*self*.model\_name = *model\_name*

*self*.output\_dir = Path(*output\_dir*)

*self*.max\_input\_length = *max\_input\_length*

*self*.max\_target\_length = *max\_target\_length*

*self*.device = torch.device("cuda" *if* torch.cuda.is\_available() *else* "cpu")

*# Create output directory*

*self*.output\_dir.mkdir(*exist\_ok*=True)

*# Initialize model and tokenizer*

*self*.tokenizer = AutoTokenizer.from\_pretrained(*model\_name*)

*self*.model = AutoModelForSeq2SeqLM.from\_pretrained(*model\_name*)

*self*.model.to(*self*.device)

*# Log device information*

        logger.info(f"Using device: {*self*.device}")

*if* torch.cuda.is\_available():

            logger.info(f"GPU: {torch.cuda.get\_device\_name(0)}")

            logger.info(f"Memory allocated: {torch.cuda.memory\_allocated(0) / 1024\*\*2:.2f} MB")

*# Initialize metric*

*self*.rouge = evaluate.load("rouge")

    def **load\_legal\_dataset**(*self*, *dataset\_name*: str = "billsum", *max\_samples*: int = 1000):

        """

        Load and prepare the billsum dataset with a limited number of samples.

        """

*try*:

*# Load the dataset*

            dataset = load\_dataset(*dataset\_name*)

            logger.info(f"Successfully loaded dataset: {*dataset\_name*}")

*# Take only max\_samples from train and a proportional amount from test*

            train\_size = min(len(dataset["train"]), *max\_samples*)

            test\_size = min(len(dataset["test"]), *max\_samples* // 5)

            dataset["train"] = dataset["train"].select(range(train\_size))

            dataset["test"] = dataset["test"].select(range(test\_size))

            logger.info(f"Using {train\_size} training samples and {test\_size} test samples")

*# Log some sample lengths before filtering*

*for* split *in* ["train", "test"]:

                sample\_idx = 0

                text\_len = len(dataset[split][sample\_idx]["text"])

                summary\_len = len(dataset[split][sample\_idx]["summary"])

                logger.info(f"Sample {split} text length: {text\_len}")

                logger.info(f"Sample {split} summary length: {summary\_len}")

*# Filter out examples where summary is too long or text is empty*

            def **filter\_examples**(*example*):

                text\_len = len(*example*["text"])

                summary\_len = len(*example*["summary"])

*# More lenient filtering criteria*

                is\_valid = (

                    text\_len >= 100 and  *# Minimum text length*

                    summary\_len >= 10 and  *# Minimum summary length*

                    text\_len <= 15000 and  *# Increased maximum length*

                    summary\_len <= 2000     *# Increased maximum length*

                )

*return* is\_valid

            filtered\_dataset = dataset.filter(

                filter\_examples,

*desc*="Filtering dataset"

            )

            filtered\_train\_size = len(filtered\_dataset["train"])

            filtered\_test\_size = len(filtered\_dataset["test"])

            logger.info(f"Dataset size after filtering: train={filtered\_train\_size}, test={filtered\_test\_size}")

*if* filtered\_train\_size == 0 or filtered\_test\_size == 0:

*raise* ValueError("All examples were filtered out. Please check the filtering criteria.")

*return* filtered\_dataset

*except* Exception *as* e:

            logger.error(f"Error loading dataset: {str(e)}")

*raise*

    def **preprocess\_data**(*self*, *examples*: Dict) -> Dict:

        """Preprocess the data for training"""

*# Tokenize inputs*

        model\_inputs = *self*.tokenizer(

*examples*["text"],

*max\_length*=*self*.max\_input\_length,

*padding*="max\_length",

*truncation*=True,

*return\_tensors*=None,  *# Ensure we don't get tensors yet*

        )

*# Tokenize targets*

*with* *self*.tokenizer.as\_target\_tokenizer():  *# Properly handle target tokenization*

            labels = *self*.tokenizer(

*examples*["summary"],

*max\_length*=*self*.max\_target\_length,

*padding*="max\_length",

*truncation*=True,

*return\_tensors*=None,  *# Ensure we don't get tensors yet*

            )

        model\_inputs["labels"] = labels["input\_ids"]

*# Replace padding token id with -100 for loss calculation*

        model\_inputs["labels"] = [

            [-100 *if* token == *self*.tokenizer.pad\_token\_id *else* token *for* token *in* label]

*for* label *in* model\_inputs["labels"]

        ]

*# Verify we have valid labels (debug)*

*if* any(all(l == -100 *for* l *in* label) *for* label *in* model\_inputs["labels"]):

            logger.warning("Found sequence with all -100 labels!")

*return* model\_inputs

    def **compute\_metrics**(*self*, *eval\_pred*) -> Dict:

        """Compute ROUGE metrics"""

        predictions, labels = *eval\_pred*

        decoded\_preds = *self*.tokenizer.batch\_decode(predictions, *skip\_special\_tokens*=True)

*# Replace -100 with pad token id*

        labels = np.where(labels != -100, labels, *self*.tokenizer.pad\_token\_id)

        decoded\_labels = *self*.tokenizer.batch\_decode(labels, *skip\_special\_tokens*=True)

        result = *self*.rouge.compute(

*predictions*=decoded\_preds,

*references*=decoded\_labels,

*use\_stemmer*=True

        )

*# Log a sample prediction*

*if* len(decoded\_preds) > 0:

            logger.info("\nSample Prediction:")

            logger.info(f"Predicted: {decoded\_preds[0][:200]}...")

            logger.info(f"Reference: {decoded\_labels[0][:200]}...")

*return* {k: round(v \* 100, 2) *for* k, v *in* result.items()}

    def **train**(

*self*,

*train\_dataset*,

*eval\_dataset*,

*batch\_size*: int = 8,

*num\_epochs*: int = 20,

*learning\_rate*: float = 3e-5

    ):

        """Train the model"""

*# Define training arguments*

        training\_args = Seq2SeqTrainingArguments(

*output\_dir*=str(*self*.output\_dir),

*num\_train\_epochs*=*num\_epochs*,

*per\_device\_train\_batch\_size*=*batch\_size*,

*per\_device\_eval\_batch\_size*=*batch\_size* \* 2,

*learning\_rate*=*learning\_rate*,

*weight\_decay*=0.01,

*logging\_dir*=str(*self*.output\_dir / "logs"),

*logging\_steps*=10,

*save\_total\_limit*=2,

*save\_steps*=50,

*eval\_steps*=50,

*gradient\_accumulation\_steps*=4,

*predict\_with\_generate*=True,

*generation\_max\_length*=*self*.max\_target\_length,

*generation\_num\_beams*=2,

*fp16*=False,

*remove\_unused\_columns*=False,

*label\_names*=["labels"],

*include\_inputs\_for\_metrics*=True,

*dataloader\_num\_workers*=0,  *# Set to 0 to debug*

*gradient\_checkpointing*=False,  *# Disable for now*

*optim*="adamw\_torch",

*warmup\_ratio*=0.1,

*max\_grad\_norm*=1.0,

*adam\_beta1*=0.9,

*adam\_beta2*=0.999,

*adam\_epsilon*=1e-8,

*lr\_scheduler\_type*="linear"  *# Correct parameter name for learning rate schedule*

        )

*# Create data collator*

        data\_collator = DataCollatorForSeq2Seq(

*tokenizer*=*self*.tokenizer,

*model*=*self*.model,

*padding*=True,

*label\_pad\_token\_id*=-100  *# Explicitly set label padding*

        )

*# Reset model weights to ensure proper initialization*

*self*.model = AutoModelForSeq2SeqLM.from\_pretrained(*self*.model\_name)

*self*.model.to(*self*.device)

*# Ensure model is in training mode*

*self*.model.train()

*# Zero gradients*

*for* param *in* *self*.model.parameters():

*if* param.requires\_grad:

                param.grad = None

*# Initialize trainer*

        trainer = Seq2SeqTrainer(

*model*=*self*.model,

*args*=training\_args,

*train\_dataset*=*train\_dataset*,

*eval\_dataset*=*eval\_dataset*,

*tokenizer*=*self*.tokenizer,

*data\_collator*=data\_collator,

*compute\_metrics*=*self*.compute\_metrics

        )

*# Train the model*

        logger.info("Starting training...")

        trainer.train()

*# Save the model*

*self*.model.save\_pretrained(*self*.output\_dir / "final\_model")

*self*.tokenizer.save\_pretrained(*self*.output\_dir / "final\_model")

        logger.info(f"Model saved to {*self*.output\_dir / 'final\_model'}")

def **main**():

*# Initialize trainer*

    trainer = LegalDocumentTrainer()

*# Load dataset*

    dataset = trainer.load\_legal\_dataset()

*# Split dataset*

    train\_dataset = dataset["train"]

    eval\_dataset = dataset["test"]

*# Preprocess datasets*

    train\_dataset = train\_dataset.map(

        trainer.preprocess\_data,

*batched*=True,

*remove\_columns*=train\_dataset.column\_names,

*desc*="Preprocessing training data"

    )

    eval\_dataset = eval\_dataset.map(

        trainer.preprocess\_data,

*batched*=True,

*remove\_columns*=eval\_dataset.column\_names,

*desc*="Preprocessing validation data"

    )

*# Train the model*

    trainer.train(train\_dataset, eval\_dataset)

*if* \_\_name\_\_ == "\_\_main\_\_":

    main()

**Model File**

*from* transformers *import* AutoTokenizer, AutoModelForSeq2SeqLM

*from* typing *import* List, Dict, Union

*import* torch

*import* nltk

*from* nltk.tokenize *import* sent\_tokenize, word\_tokenize

*from* nltk.corpus *import* stopwords

*from* nltk.cluster.util *import* cosine\_distance

*import* numpy *as* np

*import* networkx *as* nx

*import* logging

*# Configure logging*

logging.basicConfig(*level*=logging.INFO)

*# Download required NLTK resources*

def **download\_nltk\_resources**():

    """Download required NLTK resources"""

    resources = ['punkt', 'stopwords', 'averaged\_perceptron\_tagger']

*for* resource *in* resources:

*try*:

            nltk.data.find(f'tokenizers/{resource}')

            logging.info(f"Resource {resource} already downloaded")

*except* LookupError:

*try*:

                nltk.download(resource)

                logging.info(f"Successfully downloaded {resource}")

*except* Exception *as* e:

                logging.error(f"Error downloading {resource}: {str(e)}")

*# Download resources at module level*

download\_nltk\_resources()

class LegalDocumentSummarizer:

    def **\_\_init\_\_**(*self*, *model\_name*: str = "facebook/bart-large-cnn"):

        """

        Initialize the legal document summarizer.

        Args:

            model\_name (str): Name of the pre-trained model to use

        """

*self*.device = torch.device("cuda" *if* torch.cuda.is\_available() *else* "cpu")

*self*.tokenizer = AutoTokenizer.from\_pretrained(*model\_name*)

*self*.model = AutoModelForSeq2SeqLM.from\_pretrained(*model\_name*)

*self*.model.to(*self*.device)

*self*.stop\_words = set(stopwords.words('english'))

        logging.info(f"Model loaded on {*self*.device}")

    def **\_create\_sentence\_similarity\_matrix**(*self*, *sentences*: List[str]) -> np.ndarray:

        """Create similarity matrix for sentences"""

        similarity\_matrix = np.zeros((len(*sentences*), len(*sentences*)))

*for* idx1 *in* range(len(*sentences*)):

*for* idx2 *in* range(len(*sentences*)):

*if* idx1 != idx2:

                    similarity\_matrix[idx1][idx2] = *self*.\_sentence\_similarity(

*sentences*[idx1],

*sentences*[idx2]

                    )

*return* similarity\_matrix

    def **\_sentence\_similarity**(*self*, *sent1*: str, *sent2*: str) -> float:

        """Calculate similarity between two sentences"""

        words1 = [word.lower() *for* word *in* word\_tokenize(*sent1*) *if* word.lower() not in *self*.stop\_words]

        words2 = [word.lower() *for* word *in* word\_tokenize(*sent2*) *if* word.lower() not in *self*.stop\_words]

        all\_words = list(set(words1 + words2))

        vector1 = [0] \* len(all\_words)

        vector2 = [0] \* len(all\_words)

*for* word *in* words1:

            vector1[all\_words.index(word)] += 1

*for* word *in* words2:

            vector2[all\_words.index(word)] += 1

*return* 1 - cosine\_distance(vector1, vector2)

    def **generate\_extractive\_summary**(*self*, *text*: str, *num\_sentences*: int = 3) -> str:

        """

        Generate extractive summary using TextRank algorithm

        Args:

            text (str): Input text to summarize

            num\_sentences (int): Number of sentences in summary

        Returns:

            str: Extractive summary

        """

*try*:

*# Tokenize text into sentences*

            sentences = sent\_tokenize(*text*)

*if* len(sentences) <= *num\_sentences*:

*return* *text*

*# Create similarity matrix*

            similarity\_matrix = *self*.\_create\_sentence\_similarity\_matrix(sentences)

*# Create graph and calculate scores*

            graph = nx.from\_numpy\_array(similarity\_matrix)

            scores = nx.pagerank(graph)

*# Get top sentences*

            ranked\_sentences = sorted(

                ((scores[i], sentence) *for* i, sentence *in* enumerate(sentences)),

*reverse*=True

            )

*# Select top N sentences and sort by original position*

            selected\_indices = [sentences.index(sent) *for* \_, sent *in* ranked\_sentences[:*num\_sentences*]]

            selected\_indices.sort()

*# Combine sentences in original order*

            summary = ' '.join(sentences[i] *for* i *in* selected\_indices)

*return* summary

*except* Exception *as* e:

            logging.error(f"Error in extractive summarization: {str(e)}")

*raise*

    def **preprocess\_text**(*self*, *text*: str) -> List[str]:

        """

        Preprocess the input text by splitting it into chunks that fit the model's max length.

        Args:

            text (str): Input legal document text

        Returns:

            List[str]: List of text chunks

        """

*try*:

            sentences = sent\_tokenize(*text*)

*except* Exception *as* e:

            logging.error(f"Error in sentence tokenization: {str(e)}")

            sentences = [s.strip() *for* s *in* *text*.split('.') *if* s.strip()]

        chunks = []

        current\_chunk = []

        current\_length = 0

*for* sentence *in* sentences:

            sentence\_tokens = *self*.tokenizer.tokenize(sentence)

            sentence\_length = len(sentence\_tokens)

*if* current\_length + sentence\_length <= *self*.tokenizer.model\_max\_length:

                current\_chunk.append(sentence)

                current\_length += sentence\_length

*else*:

                chunks.append(" ".join(current\_chunk))

                current\_chunk = [sentence]

                current\_length = sentence\_length

*if* current\_chunk:

            chunks.append(" ".join(current\_chunk))

*return* chunks

    def **generate\_summary**(*self*, *text*: str, *max\_length*: int = 150, *min\_length*: int = 50,

*method*: str = "abstractive", *num\_sentences*: int = 3) -> Dict[str, Union[str, float]]:

        """

        Generate a summary for the given legal document.

        Args:

            text (str): Input legal document text

            max\_length (int): Maximum length of the generated summary

            min\_length (int): Minimum length of the generated summary

            method (str): Summarization method ("abstractive" or "extractive")

            num\_sentences (int): Number of sentences for extractive summary

        Returns:

            Dict[str, Union[str, float]]: Dictionary containing summary and metadata

        """

*try*:

*if* *method* == "extractive":

                summary = *self*.generate\_extractive\_summary(*text*, *num\_sentences*)

*else*:  *# abstractive*

                chunks = *self*.preprocess\_text(*text*)

                summaries = []

*for* chunk *in* chunks:

                    inputs = *self*.tokenizer(chunk, *return\_tensors*="pt", *max\_length*=1024, *truncation*=True)

                    inputs = inputs.to(*self*.device)

                    summary\_ids = *self*.model.generate(

                        inputs["input\_ids"],

*num\_beams*=4,

*max\_length*=*max\_length*,

*min\_length*=*min\_length*,

*length\_penalty*=2.0,

*early\_stopping*=True

                    )

                    summary = *self*.tokenizer.decode(summary\_ids[0], *skip\_special\_tokens*=True)

                    summaries.append(summary)

                summary = " ".join(summaries)

*return* {

                "summary": summary,

                "original\_length": len(*text*),

                "summary\_length": len(summary),

                "compression\_ratio": len(summary) / len(*text*),

                "method": *method*

            }

*except* Exception *as* e:

            logging.error(f"Error in generate\_summary: {str(e)}")

*raise*

    def **evaluate\_summary**(*self*, *original\_text*: str, *generated\_summary*: str) -> Dict[str, float]:

        """

        Evaluate the quality of the generated summary.

        Args:

            original\_text (str): Original legal document text

            generated\_summary (str): Generated summary

        Returns:

            Dict[str, float]: Dictionary containing evaluation metrics

        """

        compression\_ratio = len(*generated\_summary*) / len(*original\_text*)

*return* {

            "compression\_ratio": compression\_ratio,

        }

**App file**

*import* streamlit *as* st

*import* torch

*from* model *import* LegalDocumentSummarizer

*import* logging

*import* warnings

*import* os

*import* io

*# Configure logging*

logging.basicConfig(*level*=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

*# Suppress warnings*

warnings.filterwarnings("ignore", *category*=UserWarning)

warnings.filterwarnings("ignore", *category*=FutureWarning)

*# Try importing optional dependencies with explicit logging*

*try*:

*import* PyPDF2

    logger.info("Successfully loaded PyPDF2")

    PDF\_AVAILABLE = True

*except* ImportError *as* e:

    logger.warning(f"PyPDF2 import failed: {str(e)}")

    PDF\_AVAILABLE = False

*try*:

*import* chardet

    logger.info("Successfully loaded chardet")

    CHARDET\_AVAILABLE = True

*except* ImportError *as* e:

    logger.warning(f"chardet import failed: {str(e)}")

    CHARDET\_AVAILABLE = False

*try*:

*import* docx2txt

    logger.info("Successfully loaded docx2txt")

    DOCX\_AVAILABLE = True

*except* ImportError *as* e:

    logger.warning(f"docx2txt import failed: {str(e)}")

    DOCX\_AVAILABLE = False

def **read\_file\_content**(*uploaded\_file*):

    """Read content from different file types with encoding detection"""

*try*:

*# Get file extension*

        file\_type = *uploaded\_file*.name.split('.')[-1].lower()

        logger.info(f"Processing file of type: {file\_type}")

*if* file\_type == 'txt':

*# Read the file content as bytes first*

            content\_bytes = *uploaded\_file*.read()

*if* CHARDET\_AVAILABLE:

*# Detect the encoding*

                result = chardet.detect(content\_bytes)

                encoding = result['encoding'] *if* result['encoding'] *else* 'utf-8'

                logger.info(f"Detected encoding: {encoding}")

*else*:

*# If chardet is not available, try common encodings*

                encoding = 'utf-8'

                logger.info("Using default utf-8 encoding")

*try*:

*# Try detected encoding*

*return* content\_bytes.decode(encoding)

*except* UnicodeDecodeError:

*# Fallback encodings*

*for* enc *in* ['utf-8', 'latin-1', 'cp1252', 'iso-8859-1']:

*try*:

*return* content\_bytes.decode(enc)

*except* UnicodeDecodeError:

*continue*

*raise* UnicodeDecodeError(f"Could not decode file with any common encoding")

*elif* file\_type == 'docx':

*if* not DOCX\_AVAILABLE:

*raise* ImportError("docx2txt package is not installed. Please install it to read .docx files.")

*# Extract text from docx*

            text = docx2txt.process(*uploaded\_file*)

*return* text

*elif* file\_type == 'pdf':

*if* not PDF\_AVAILABLE:

*raise* ImportError("PyPDF2 package is not installed. Please install it to read .pdf files.")

*# Read PDF file*

*try*:

                pdf\_bytes = io.BytesIO(*uploaded\_file*.getvalue())

                pdf\_reader = PyPDF2.PdfReader(pdf\_bytes)

                text = ""

*for* page *in* pdf\_reader.pages:

                    text += page.extract\_text() + "\n"

                logger.info(f"Successfully extracted text from PDF with {len(pdf\_reader.pages)} pages")

*return* text

*except* Exception *as* e:

                logger.error(f"Error processing PDF: {str(e)}")

*raise*

*else*:

*raise* ValueError(f"Unsupported file type: {file\_type}")

*except* Exception *as* e:

        logger.error(f"Error reading file: {str(e)}")

*raise*

*# Set page config*

st.set\_page\_config(

*page\_title*="Legal Document Summarizer",

*page\_icon*="⚖️",

*layout*="wide"

)

@st.cache\_resource(*show\_spinner*=True)

def **load\_model**():

    """Load the model and return an instance of LegalDocumentSummarizer"""

*try*:

*# Check if fine-tuned model exists*

        model\_path = "fine\_tuned\_model/final\_model"

*if* os.path.exists(model\_path):

*return* LegalDocumentSummarizer(*model\_name*=model\_path)

*else*:

*return* LegalDocumentSummarizer()

*except* Exception *as* e:

        st.error(f"Error loading model: {str(e)}")

        logging.error(f"Model loading error: {str(e)}")

*return* None

def **main**():

    st.title("Legal Document Summarizer")

    st.write("Upload or paste your legal document to generate a summary.")

*# Initialize model*

    model = load\_model()

*if* model is None:

        st.error("Failed to load the model. Please try again.")

*return*

*# Show supported file types based on available packages*

    supported\_types = ["txt"]

*if* DOCX\_AVAILABLE:

        supported\_types.append("docx")

*if* PDF\_AVAILABLE:

        supported\_types.append("pdf")

    logger.info(f"Available file types: {supported\_types}")

*# Input methods*

    input\_method = st.radio("Choose input method:", ["Upload File", "Paste Text"])

    input\_text = ""

*if* input\_method == "Upload File":

        st.info(f"Supported file types: {', '.join(supported\_types)}")

        uploaded\_file = st.file\_uploader("Choose a file", *type*=supported\_types)

*if* uploaded\_file:

*try*:

*with* st.spinner(f"Reading {uploaded\_file.name}..."):

                    input\_text = read\_file\_content(uploaded\_file)

                st.success(f"Successfully read file: {uploaded\_file.name}")

*except* ImportError *as* e:

                st.error(str(e))

                st.info("Please install the required package or use the paste text option.")

*except* Exception *as* e:

                st.error(f"Error reading file: {str(e)}")

                st.info("Try uploading a different file or use the paste text option.")

*else*:

        input\_text = st.text\_area("Paste your legal document here:", *height*=300)

*# Summarization parameters*

    st.subheader("Summarization Settings")

*# Method selection*

    method = st.radio(

        "Choose summarization method:",

        ["Abstractive", "Extractive"],

*help*="Abstractive: Generates new text. Extractive: Selects important sentences from the original text."

    )

    col1, col2 = st.columns(2)

*if* method == "Abstractive":

*with* col1:

            max\_length = st.slider("Maximum summary length", 50, 500, 150)

*with* col2:

            min\_length = st.slider("Minimum summary length", 30, 200, 50)

*else*:  *# Extractive*

*with* col1:

            num\_sentences = st.slider("Number of sentences to extract", 3, 10, 5)

            max\_length = 500  *# Default for extractive*

            min\_length = 50   *# Default for extractive*

*if* st.button("Generate Summary") and input\_text:

*if* len(input\_text.strip()) < 100:

            st.warning("The input text is too short. Please provide a longer document.")

*return*

*with* st.spinner("Generating summary..."):

*try*:

                result = model.generate\_summary(

                    input\_text,

*max\_length*=max\_length,

*min\_length*=min\_length,

*method*=method.lower(),

*num\_sentences*=num\_sentences *if* method == "Extractive" *else* 3

                )

                st.subheader("Generated Summary")

                st.write(result["summary"])

                st.subheader("Statistics")

                col1, col2, col3, col4 = st.columns(4)

*with* col1:

                    st.metric("Original Length", f"{result['original\_length']} chars")

*with* col2:

                    st.metric("Summary Length", f"{result['summary\_length']} chars")

*with* col3:

                    st.metric("Compression Ratio", f"{result['compression\_ratio']:.2%}")

*with* col4:

                    st.metric("Method", result["method"].title())

*except* Exception *as* e:

                st.error(f"Error generating summary: {str(e)}")

                logging.error(f"Summary generation error: {str(e)}")

    st.markdown("---")

    st.markdown("""

    ### Tips

    - For best results, ensure your document is well-formatted

    - The model works best with documents between 1,000 and 15,000 characters

    - Abstractive summarization generates new text, while extractive selects important sentences

    - Extractive summarization is faster but less flexible

    - Currently supported file formats: {', '.join(supported\_types)}

    """)

*if* \_\_name\_\_ == "\_\_main\_\_":

    main()