CODE

app.py

```
import os
import datetime
import nltk
import time
import pandas as pd
import streamlit as st
from summarizer import TextSummarizer
import database as db

# Explicitly set the data path for NLTK
nltk.data.path.append('/Users/suryakandikattu/nltk_data')

# Ensure necessary NLTK resources are downloaded, explicitly download punkt
try:
    nltk.data.find('tokenizers/punkt')
    print("Punkt resource found!")
except LookupError:
    print("Punkt resource not found. Downloading...")
    nltk.download('punkt')

# Explicitly download 'averaged_perceptron_tagger'
try:
    nltk.data.find('taggers/averaged_perceptron_tagger')
    print("Averaged Perceptron Tagger found!")
except LookupError:
```

```
    print("Averaged Perceptron Tagger not found. Downloading...")
    nltk.download('averaged_perceptron_tagger')

# Set page configuration
st.set_page_config(
    page_title="Text Summarizer",
    page_icon="",
    layout="wide"
)

# Initialize the summarizer
summarizer = TextSummarizer()

def main():
    # Create sidebar navigation
    page = st.sidebar.selectbox("Navigation", ["New Summary", "History"])

    if page == "New Summary":
        create_summary_page()
    else:
        view_history_page()

def create_summary_page():
    st.title("NLP Text Summarization")
    st.write("Upload or paste text to generate a concise summary using NLP techniques.")

    # File upload option
```

```
    uploaded_file = st.file_uploader("Upload a text file", type=["txt", "csv", "md"])

    # Text input option
    text_input = st.text_area("Or paste your text here:", height=250)

    # Add title field for the summary
    summary_title = st.text_input("Summary Title:", "Untitled Summary")

    # Parameters for summarization
    st.sidebar.header("Summarization Parameters")

    # Summary length parameter (sentence count or percentage)
    length_option = st.sidebar.radio("Summary Length Type:", ["Sentence Count", "Percentage"])

    if length_option == "Sentence Count":
        summary_length = st.sidebar.slider("Number of sentences:", min_value=1, max_value=20, value=5)
        summary_percentage = None
    else:
        summary_percentage = st.sidebar.slider("Summary Percentage:", min_value=10, max_value=90, value=30)
        summary_length = None

    # Algorithm selection
    algorithm = st.sidebar.selectbox(
        "Summarization Algorithm:",
        ["TextRank", "Frequency-based", "Latent Semantic Analysis"]
    )
```

```
# Focus parameter (what type of sentences to prioritize)
focus = st.sidebar.multiselect(
    "Focus on:",
    ["Key entities", "Action sentences", "Quotations", "Statistics"],
    default=["Key entities"]
)

# Save to database checkbox
save_to_db = st.sidebar.checkbox("Save summary to database", value=True)

# Process text when submitted
if st.button("Generate Summary"):
    # Get the text content
    text_content = ""

    if uploaded_file is not None:
        try:
            text_content = uploaded_file.getvalue().decode("utf-8")
        except UnicodeDecodeError:
            st.error("Unable to decode the file. Please ensure it's a valid text file.")
            return
    elif text_input:
        text_content = text_input
    else:
        st.warning("Please upload a file or paste text to summarize.")
        return
```

```
if text_content.strip():
    # Show progress
    progress_bar = st.progress(0)
    status_text = st.empty()

    # Phase 1: Text Processing
    status_text.text("Processing text...")
    progress_bar.progress(25)
    time.sleep(0.5)  # Simulating processing time

    # Phase 2: Analysis
    status_text.text("Analyzing content...")
    progress_bar.progress(50)
    time.sleep(0.5)  # Simulating processing time

    # Phase 3: Generating Summary
    status_text.text("Generating summary...")
    progress_bar.progress(75)

    try:
        # Generate summary using the TextSummarizer
        summary = summarizer.summarize(
            text_content,
            algorithm=algorithm,
            sentence_count=summary_length,
            percentage=summary_percentage,
```

```
    focus=focus
)

# Calculate metrics (compression ratio, etc.)
metrics = summarizer.get_metrics(text_content, summary)
compression_ratio = metrics["Compression Ratio (%)"]

# Save to database if checkbox is selected
if save_to_db:
    saved_summary = db.save_summary(
        original_text=text_content,
        summary_text=summary,
        algorithm=algorithm,
        sentence_count=summary_length,
        percentage=summary_percentage,
        focus=focus,
        compression_ratio=compression_ratio,
        title=summary_title
    )
    status_text.text(f"Summary generated and saved with ID: {saved_summary.id}")
else:
    status_text.text("Summary generated!")

# Complete the progress
progress_bar.progress(100)
time.sleep(0.5)  # Give user time to see the completion
```

```
# Clear progress indicators
progress_bar.empty()
status_text.empty()

# Display results
col1, col2 = st.columns(2)

with col1:
    st.subheader("Original Text")
    st.text_area("", value=text_content, height=400, disabled=True)
    word_count = len(text_content.split())
    st.write(f"Word count: {word_count}")

with col2:
    st.subheader("Summary")
    st.text_area("", value=summary, height=400, disabled=True)
    summary_word_count = len(summary.split())
    st.write(f"Word count: {summary_word_count}")
    st.write(f"Compression ratio: {compression_ratio}%")

# Display key metrics
st.subheader("Summary Metrics")

# Display metrics in a dataframe for better visualization
metrics_df = pd.DataFrame([metrics])
st.dataframe(metrics_df)
```

```
            except Exception as e:
                st.error(f"An error occurred during summarization: {str(e)}")
        else:
            st.warning("The provided text is empty. Please provide some content to summarize.")

def view_history_page():
    st.title("Summary History")

    # Get all summaries from the database
    summaries = db.get_summaries()

    if not summaries:
        st.info("No summaries found in the database.")
        return

    # Create a dataframe for viewing the summaries
    summary_list = []
    for s in summaries:
        # Format the datetime
        created_at = s.created_at.strftime("%Y-%m-%d %H:%M")

        # Truncate the summary text for display
        short_summary = s.summary_text[:100] + "..." if len(s.summary_text) > 100 else
s.summary_text

        summary_list.append({
            "ID": s.id,
            "Title": s.title,
```

```
        "Algorithm": s.algorithm,
        "Created": created_at,
        "Compression": f"{s.compression_ratio:.2f}%",
        "Preview": short_summary,
        "Favorite": "✓" if s.is_favorite else ""
    })

# Display summaries as a dataframe
summary_df = pd.DataFrame(summary_list)
st.dataframe(summary_df, use_container_width=True)

# View full summary
col1, col2 = st.columns(2)

with col1:
    selected_id = st.number_input("Enter Summary ID to view:", min_value=1,
                                  max_value=max([s.id for s in summaries]) if summaries else 1,
                                  step=1)

    if st.button("View Full Summary"):
        selected_summary = db.get_summary(selected_id)
        if selected_summary:
            st.subheader(f"Summary: {selected_summary.title}")

            # Display metadata
            meta_col1, meta_col2 = st.columns(2)
            with meta_col1:
```

```
            st.write(f"**Algorithm:** {selected_summary.algorithm}")
            st.write(f"**Created:** {selected_summary.created_at.strftime('%Y-%m-%d %H:%M')}")
        with meta_col2:
            st.write(f"**Compression Ratio:** {selected_summary.compression_ratio:.2f}%")
            if selected_summary.focus:
                st.write(f"**Focus:** {selected_summary.focus}")

        # Display text areas
        text_col1, text_col2 = st.columns(2)
        with text_col1:
            st.write("**Original Text:**")
            st.text_area("", value=selected_summary.original_text, height=400, disabled=True)

        with text_col2:
            st.write("**Summary:**")
            st.text_area("", value=selected_summary.summary_text, height=400, disabled=True)
    else:
        st.error("Summary not found.")

if __name__ == "__main__":
    main()
```

database.py

```
import os
import datetime
from sqlalchemy import create_engine, Column, Integer, String, Text, DateTime, Float, Boolean
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# Get database URL from environment variable or default to SQLite if not set
DATABASE_URL = os.environ.get("DATABASE_URL", "sqlite:///./summaries.db")

# Create SQLAlchemy engine and session
engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False} if "sqlite" in DATABASE_URL else {})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Create base class for declarative models
Base = declarative_base()

# Define Summary model
class Summary(Base):
    __tablename__ = "summaries"

    id = Column(Integer, primary_key=True, index=True)
    original_text = Column(Text)
    summary_text = Column(Text)
    algorithm = Column(String(50))
    sentence_count = Column(Integer, nullable=True)
    percentage = Column(Integer, nullable=True)
```

```
    focus = Column(String(255), nullable=True)
    compression_ratio = Column(Float)
    created_at = Column(DateTime, default=datetime.datetime.utcnow)

    # We'll add a title field that can be used to easily identify the summary
    title = Column(String(255))

    # We'll also add a favorite field
    is_favorite = Column(Boolean, default=False)

# Create the tables in the database (only if they don't exist)
Base.metadata.create_all(bind=engine)

# Database operations
def get_db():
    """
    Get a database session.

    This function ensures that a session is created and managed properly for each transaction.
    """
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

def save_summary(original_text, summary_text, algorithm, sentence_count=None,
```

```
        percentage=None, focus=None, compression_ratio=0.0, title="Untitled Summary"):
"""
Save a summary to the database.

Args:
    original_text (str): The original text
    summary_text (str): The generated summary
    algorithm (str): The algorithm used for summarization
    sentence_count (int): Number of sentences in the summary
    percentage (int): Percentage of original text to keep
    focus (list): Features prioritized in the summary
    compression_ratio (float): The compression ratio
    title (str): A title for the summary

Returns:
    Summary: The saved summary object
"""
db = next(get_db())  # Fetch the database session using the context manager

# Convert focus list to string if present
focus_str = None
if focus and isinstance(focus, list):
    focus_str = ", ".join(focus)

# Create new summary object
new_summary = Summary(
    original_text=original_text,
```

```
        summary_text=summary_text,
        algorithm=algorithm,
        sentence_count=sentence_count,
        percentage=percentage,
        focus=focus_str,
        compression_ratio=compression_ratio,
        title=title
    )

    # Add and commit to the database
    db.add(new_summary)
    db.commit()
    db.refresh(new_summary)
    return new_summary

def get_summaries(limit=10, skip=0):
    """
    Get all summaries from the database.

    Args:
        limit (int): Maximum number of summaries to return
        skip (int): Number of summaries to skip

    Returns:
        list: List of Summary objects
    """
    db = next(get_db())  # Fetch the database session using the context manager
```

```
    return db.query(Summary).order_by(Summary.created_at.desc()).offset(skip).limit(limit).all()

def get_summary(summary_id):
    """
    Get a summary by ID.

    Args:
        summary_id (int): The ID of the summary

    Returns:
        Summary: The summary object or None if not found
    """
    db = next(get_db())  # Fetch the database session using the context manager
    return db.query(Summary).filter(Summary.id == summary_id).first()

def update_summary(summary_id, title=None, is_favorite=None):
    """
    Update a summary's title or favorite status.

    Args:
        summary_id (int): The ID of the summary
        title (str): The new title for the summary
        is_favorite (bool): Whether the summary is a favorite

    Returns:
        Summary: The updated summary object or None if not found
    """
```

```
    db = next(get_db())  # Fetch the database session using the context manager
    summary = db.query(Summary).filter(Summary.id == summary_id).first()

    if not summary:
        return None

    if title is not None:
        summary.title = title

    if is_favorite is not None:
        summary.is_favorite = is_favorite

    db.commit()
    db.refresh(summary)
    return summary

def delete_summary(summary_id):
    """
    Delete a summary by ID.

    Args:
        summary_id (int): The ID of the summary

    Returns:
        bool: True if summary was deleted, False otherwise
    """
    db = next(get_db())  # Fetch the database session using the context manager
```

```
summary = db.query(Summary).filter(Summary.id == summary_id).first()


if not summary:
    return False


db.delete(summary)
db.commit()
return True
```

summarizer.py

```
import nltk
import numpy as np
import networkx as nx
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.decomposition import TruncatedSVD
from text_processor import TextProcessor

# Download necessary NLTK resources
try:
    nltk.data.find('tokenizers/punkt')
    nltk.data.find('stopwords')
except LookupError:
    nltk.download('punkt')
    nltk.download('stopwords')

class TextSummarizer:
    def __init__(self):
        self.text_processor = TextProcessor()

    def summarize(self, text, algorithm="TextRank", sentence_count=None, percentage=None,
focus=None):
        """
        Generate a summary of the given text using the specified algorithm.

        Args:
            text (str): The input text to summarize
            algorithm (str): The summarization algorithm to use
```

```
    sentence_count (int): Number of sentences in the summary
    percentage (int): Percentage of original text to keep
    focus (list): Features to prioritize in the summary

Returns:
    str: The summarized text
"""
# Validate input
if not text.strip():
    return ""

# Preprocess the text
sentences = self.text_processor.get_sentences(text)

# If there are no sentences after preprocessing, return empty string
if not sentences:
    return ""

# Determine the number of sentences to include in the summary
total_sentences = len(sentences)
if sentence_count:
    num_sentences = min(sentence_count, total_sentences)
elif percentage:
    num_sentences = max(1, int(total_sentences * percentage / 100))
else:
    # Default to 30% if neither is specified
    num_sentences = max(1, int(total_sentences * 0.3))
```

```
    # Select the appropriate algorithm
    if algorithm == "TextRank":
        summary_indices = self._textrank_summarize(sentences, num_sentences, focus)
    elif algorithm == "Frequency-based":
        summary_indices = self._frequency_summarize(sentences, num_sentences, focus)
    elif algorithm == "Latent Semantic Analysis":
        summary_indices = self._lsa_summarize(sentences, num_sentences, focus)
    else:
        # Default to TextRank
        summary_indices = self._textrank_summarize(sentences, num_sentences, focus)

    # Create summary by joining the selected sentences
    summary = " ".join([sentences[i] for i in sorted(summary_indices)])

    return summary

def _textrank_summarize(self, sentences, num_sentences, focus=None):
    """
    Implement TextRank algorithm for extractive summarization.

    Args:
        sentences (list): List of preprocessed sentences
        num_sentences (int): Number of sentences to include in summary
        focus (list): Features to prioritize

    Returns:
```

```
        list: Indices of sentences to include in the summary
    """
    # Create similarity matrix
    similarity_matrix = self._build_similarity_matrix(sentences)

    # Apply PageRank algorithm
    scores = nx.pagerank(nx.from_numpy_array(similarity_matrix))

    # Adjust scores based on focus if specified
    if focus:
        scores = self._adjust_scores_by_focus(sentences, scores, focus)

    # Sort sentences by score and return top indices
    ranked_indices = sorted(((scores[i], i) for i in range(len(sentences))), reverse=True)
    return [ranked_indices[i][1] for i in range(min(num_sentences, len(ranked_indices)))]

def _frequency_summarize(self, sentences, num_sentences, focus=None):
    """
    Implement frequency-based summarization.

    Args:
        sentences (list): List of preprocessed sentences
        num_sentences (int): Number of sentences to include in summary
        focus (list): Features to prioritize

    Returns:
        list: Indices of sentences to include in the summary
```

```
        """
        # Create a TF-IDF matrix
        vectorizer = TfidfVectorizer(stop_words='english')
        tfidf_matrix = vectorizer.fit_transform([self.text_processor.preprocess_text(sentence) for sentence in sentences])

        # Calculate sentence scores based on the sum of TF-IDF values
        sentence_scores = [sum(tfidf_matrix[i].toarray()[0]) for i in range(len(sentences))]

        # Adjust scores based on focus if specified
        if focus:
            sentence_scores = self._adjust_scores_by_focus(sentences, sentence_scores, focus)

        # Sort sentences by score and return top indices
        ranked_indices = sorted(((sentence_scores[i], i) for i in range(len(sentences))), reverse=True)
        return [ranked_indices[i][1] for i in range(min(num_sentences, len(ranked_indices)))]

    def _lsa_summarize(self, sentences, num_sentences, focus=None):
        """
        Implement Latent Semantic Analysis for summarization.

        Args:
            sentences (list): List of preprocessed sentences
            num_sentences (int): Number of sentences to include in summary
            focus (list): Features to prioritize

        Returns:
            list: Indices of sentences to include in the summary
```

```
        """
        # Create a document-term matrix
        vectorizer = CountVectorizer(stop_words='english')
        dtm = vectorizer.fit_transform([self.text_processor.preprocess_text(sentence) for sentence in sentences])

        # Apply SVD
        lsa = TruncatedSVD(n_components=min(len(sentences), 10), random_state=42)
        lsa.fit(dtm)

        # Get sentence scores
        terms_topics = lsa.components_
        sentence_scores = []

        for i, sentence in enumerate(sentences):
            sentence_vector = dtm[i].toarray()[0]
            score = 0
            for j, term_weight in enumerate(sentence_vector):
                if term_weight > 0:
                    # Sum the term weights from the most important topics
                    score += sum(abs(terms_topics[topic_idx, j]) for topic_idx in range(lsa.n_components))
            sentence_scores.append(score)

        # Adjust scores based on focus if specified
        if focus:
            sentence_scores = self._adjust_scores_by_focus(sentences, sentence_scores, focus)

        # Sort sentences by score and return top indices
```

```
        ranked_indices = sorted(((sentence_scores[i], i) for i in range(len(sentences))), reverse=True)
        return [ranked_indices[i][1] for i in range(min(num_sentences, len(ranked_indices)))]

    def _build_similarity_matrix(self, sentences):
        """
        Build a similarity matrix for the sentences.

        Args:
            sentences (list): List of sentences

        Returns:
            numpy.ndarray: Similarity matrix
        """
        # Number of sentences
        n = len(sentences)

        # Initialize similarity matrix
        similarity_matrix = np.zeros((n, n))

        # Process sentences to get vectors
        vectorizer = TfidfVectorizer(stop_words='english')
        vectors = vectorizer.fit_transform([self.text_processor.preprocess_text(sentence) for sentence in sentences])

        # Compute similarity between sentence pairs
        for i in range(n):
            for j in range(n):
                if i != j:
```

```
            similarity_matrix[i][j] = self._cosine_similarity(vectors[i], vectors[j])

    return similarity_matrix

def _cosine_similarity(self, vec1, vec2):
    """
    Calculate cosine similarity between two vectors.

    Args:
        vec1, vec2: Sparse vectors from TfidfVectorizer

    Returns:
        float: Cosine similarity value
    """
    vec1_array = vec1.toarray()[0]
    vec2_array = vec2.toarray()[0]

    dot_product = np.dot(vec1_array, vec2_array)
    norm_vec1 = np.linalg.norm(vec1_array)
    norm_vec2 = np.linalg.norm(vec2_array)

    if norm_vec1 == 0 or norm_vec2 == 0:
        return 0

    return dot_product / (norm_vec1 * norm_vec2)

def _adjust_scores_by_focus(self, sentences, scores, focus):
```

```
"""
Adjust sentence scores based on focus parameters.

Args:
    sentences (list): List of sentences
    scores (list/dict): Current sentence scores
    focus (list): Features to prioritize

Returns:
    list: Adjusted sentence scores
"""
# Convert dict to list if necessary
if isinstance(scores, dict):
    score_list = [scores[i] for i in range(len(sentences))]
else:
    score_list = scores.copy()

# Apply adjustments based on focus
for i, sentence in enumerate(sentences):
    # Focus on key entities (proper nouns, important terms)
    if "Key entities" in focus and self.text_processor.contains_key_entities(sentence):
        score_list[i] *= 1.5

    # Focus on action sentences (usually contain verbs)
    if "Action sentences" in focus and self.text_processor.is_action_sentence(sentence):
        score_list[i] *= 1.3
```

```
        # Focus on quotations
        if "Quotations" in focus and self.text_processor.contains_quotation(sentence):
            score_list[i] *= 1.4

        # Focus on statistics (sentences with numbers)
        if "Statistics" in focus and self.text_processor.contains_statistics(sentence):
            score_list[i] *= 1.4

    return score_list

def get_metrics(self, original_text, summary):
    """
    Calculate metrics for the summary.

    Args:
        original_text (str): The original text
        summary (str): The generated summary

    Returns:
        dict: Dictionary of metrics
    """
    # Calculate basic metrics
    original_word_count = len(original_text.split())
    summary_word_count = len(summary.split())
    compression_ratio = round((1 - summary_word_count/original_word_count) * 100, 2)

    # Get sentence counts
```

```
original_sentence_count = len(self.text_processor.get_sentences(original_text))
summary_sentence_count = len(self.text_processor.get_sentences(summary))

return {
    "Original Word Count": original_word_count,
    "Summary Word Count": summary_word_count,
    "Compression Ratio (%)": compression_ratio,
    "Original Sentence Count": original_sentence_count,
    "Summary Sentence Count": summary_sentence_count
}
```

textprocessor.py

```
import re
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords

# Download necessary NLTK resources
try:
    nltk.data.find('tokenizers/punkt')
    nltk.data.find('stopwords')
    nltk.data.find('taggers/averaged_perceptron_tagger')
except LookupError:
    nltk.download('punkt')
    nltk.download('stopwords')
    nltk.download('averaged_perceptron_tagger')

class TextProcessor:
    def __init__(self):
        self.stop_words = set(stopwords.words('english'))

    def get_sentences(self, text):
        """
        Split text into sentences.

        Args:
            text (str): Input text
```

```
    Returns:
        list: List of sentences
    """
    # Remove excessive whitespace
    text = re.sub(r'\s+', ' ', text).strip()

    # Split into sentences
    sentences = sent_tokenize(text)

    # Filter out very short sentences (likely not complete thoughts)
    return [s.strip() for s in sentences if len(s.split()) > 3]

def preprocess_text(self, text):
    """
    Preprocess text for analysis.

    Args:
        text (str): Input text

    Returns:
        str: Preprocessed text
    """
    # Convert to lowercase
    text = text.lower()

    # Remove special characters and numbers
    text = re.sub(r'[^\w\s]', '', text)
```

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

    # Tokenize
    words = word_tokenize(text)

    # Remove stopwords
    filtered_words = [word for word in words if word not in self.stop_words]

    # Join back to string
    return ' '.join(filtered_words)

def contains_key_entities(self, sentence):
    """
    Check if the sentence contains key entities (proper nouns or important terms).

    Args:
        sentence (str): Input sentence

    Returns:
        bool: True if sentence contains key entities
    """
    # Tokenize and POS tag
    tokens = word_tokenize(sentence)
    pos_tags = nltk.pos_tag(tokens)

    # Check for proper nouns (NNP, NNPS) or key terms
    for word, tag in pos_tags:
```

```
            if tag in ['NNP', 'NNPS']:
                return True

        # Check for capitalized words that aren't at the beginning
        for i, token in enumerate(tokens):
            if i > 0 and token[0].isupper():
                return True

        return False

    def is_action_sentence(self, sentence):
        """
        Check if the sentence describes an action (contains verbs).

        Args:
            sentence (str): Input sentence

        Returns:
            bool: True if sentence describes an action
        """
        # Tokenize and POS tag
        tokens = word_tokenize(sentence)
        pos_tags = nltk.pos_tag(tokens)

        # Check for verbs
        verb_tags = ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']
        for _, tag in pos_tags:
```

```
        if tag in verb_tags:
            return True

    return False

def contains_quotation(self, sentence):
    """
    Check if the sentence contains a quotation.

    Args:
        sentence (str): Input sentence

    Returns:
        bool: True if sentence contains a quotation
    """
    # Check for quotation marks
    quotation_pattern = re.compile(r'["\'].+?["\']')
    return bool(quotation_pattern.search(sentence))

def contains_statistics(self, sentence):
    """
    Check if the sentence contains statistics (numbers, percentages).

    Args:
        sentence (str): Input sentence

    Returns:
```

```
    bool: True if sentence contains statistics
"""
# Check for numbers and percentages
number_pattern = re.compile(r'\d+(\.\d+)?%?')
return bool(number_pattern.search(sentence))
```