import nltk

import spacy

import numpy as np

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

from nltk.corpus import stopwords

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.ensemble import RandomForestRegressor

from sklearn.model\_selection import train\_test\_split

import pandas as pd

# Load and train model with IELTS dataset

def train\_model():

    try:

        dataset = pd.read\_csv('attached\_assets/ielts\_writing\_dataset.csv')

        essays = dataset['Essay'].tolist()

        scores = dataset['Overall'].tolist()

        # Convert text to features using TF-IDF with better parameters

        vectorizer = TfidfVectorizer(

            max\_features=2000,

            min\_df=2,

            max\_df=0.95,

            ngram\_range=(1, 2)

        )

        X = vectorizer.fit\_transform(essays)

        # Split data with stratification

        X\_train, X\_test, y\_train, y\_test = train\_test\_split(

            X, scores, test\_size=0.2, random\_state=42

        )

        # Train model with optimized parameters

        model = RandomForestRegressor(

            n\_estimators=100,

            max\_depth=20,

            min\_samples\_split=5,

            random\_state=42

        )

        model.fit(X\_train, y\_train)

        # Calculate and print detailed validation metrics

        train\_score = model.score(X\_train, y\_train)

        test\_score = model.score(X\_test, y\_test)

        train\_pred = model.predict(X\_train)

        test\_pred = model.predict(X\_test)

        print("\nModel Training Metrics:")

        print(f"Training R² score: {train\_score:.3f}")

        print(f"Testing R² score: {test\_score:.3f}")

        print(f"Training MAE: {np.mean(np.abs(train\_pred - y\_train)):.3f}")

        print(f"Testing MAE: {np.mean(np.abs(test\_pred - y\_test)):.3f}")

        return model, vectorizer

    except Exception as e:

        print(f"Error during model training: {e}")

        return None, None

# Initialize model and vectorizer

MODEL, VECTORIZER = train\_model()

# Download required NLTK data

for package in ['punkt', 'stopwords', 'averaged\_perceptron\_tagger', 'punkt\_tab']:

    try:

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

    except LookupError:

        nltk.download(package)

nlp = spacy.load('en\_core\_web\_sm')

class EssayAnalyzer:

    def \_\_init\_\_(self):

        self.grammar\_checker = GrammarChecker()

        self.structure\_analyzer = StructureAnalyzer()

        self.coherence\_analyzer = CoherenceAnalyzer()

        self.vocabulary\_analyzer = VocabularyAnalyzer()

        self.scorer = EssayScorer()

class GrammarChecker:

    def check(self, text):

        doc = nlp(text)

        errors = []

        for sent in doc.sents:

            for token in sent:

                if token.dep\_ == 'nsubj' and token.head.pos\_ == 'VERB':

                    if token.tag\_ == 'NNS' and token.head.tag\_ == 'VBZ':

                        errors.append(f"Grammar error: '{token.text} {token.head.text}'")

        return errors

class StructureAnalyzer:

    def analyze(self, text):

        sentences = sent\_tokenize(text)

        return {

            'total\_sentences': len(sentences),

            'avg\_length': sum(len(sent.split()) for sent in sentences) / len(sentences),

            'sentence\_types': self.classify\_sentences(text)

        }

    def classify\_sentences(self, text):

        doc = nlp(text)

        return {

            'simple': sum(1 for sent in doc.sents if len(list(sent.root.children)) <= 1),

            'complex': sum(1 for sent in doc.sents if len(list(sent.root.children)) > 1)

        }

class CoherenceAnalyzer:

    def analyze(self, text):

        doc = nlp(text)

        return {

            'transition\_words': self.count\_transitions(doc),

            'topic\_consistency': self.check\_topic\_consistency(doc),

            'paragraph\_structure': text.count('\n\n')

        }

    def count\_transitions(self, doc):

        transitions = [

            'however', 'therefore', 'furthermore', 'moreover', 'although',

            'consequently', 'meanwhile', 'nevertheless', 'thus', 'hence',

            'additionally', 'finally', 'indeed', 'in conclusion', 'for example',

            'in fact', 'as a result', 'in addition', 'on the other hand'

        ]

        transition\_count = 0

        text\_lower = doc.text.lower()

        for transition in transitions:

            if transition in text\_lower:

                transition\_count += 1

        return transition\_count

    def check\_topic\_consistency(self, doc):

        # Get main topics (nouns and their frequencies)

        topics = {}

        for token in doc:

            if token.pos\_ == 'NOUN':

                topics[token.text.lower()] = topics.get(token.text.lower(), 0) + 1

        # Calculate topic consistency score

        if not topics:

            return 0

        return len([freq for freq in topics.values() if freq > 1])

class VocabularyAnalyzer:

    def analyze(self, text):

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

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

        content\_words = [word for word in words if word not in stop\_words and word.isalnum()]

        # Count word frequencies

        word\_freq = {}

        for word in content\_words:

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

        # Get top 10 most used words

        most\_used = sorted(word\_freq.items(), key=lambda x: x[1], reverse=True)[:10]

        return {

            'unique\_words': len(set(content\_words)),

            'total\_words': len(content\_words),

            'advanced\_words': sum(1 for word in content\_words if len(word) > 8),

            'vocabulary\_richness': len(set(content\_words)) / len(content\_words) if content\_words else 0,

            'most\_used\_words': most\_used

        }

class EssayScorer:

    def calculate\_score(self, text, analysis\_results):

        # Get model prediction

        features = VECTORIZER.transform([text])

        predicted\_score = MODEL.predict(features)[0]

        # Combine with rule-based scores

        scores = {

            'grammar': 10 - len(analysis\_results['grammar']) \* 0.5,

            'structure': min(10, analysis\_results['structure']['avg\_length'] / 5),

            'coherence': min(10, (

                analysis\_results['coherence']['transition\_words'] \* 1.5 +

                analysis\_results['coherence']['topic\_consistency'] \* 0.8 +

                analysis\_results['coherence']['paragraph\_structure'] \* 1.0

            )),

            'vocabulary': min(10, analysis\_results['vocabulary']['vocabulary\_richness'] \* 10),

            'model\_predicted': predicted\_score

        }

        weights = {'grammar': 0.2, 'structure': 0.15, 'coherence': 0.2,

                  'vocabulary': 0.2, 'model\_predicted': 0.25}

        scores['overall'] = sum(score \* weights[component] for component, score in scores.items())

        return scores

def analyze\_essay(text):

    analyzer = EssayAnalyzer()

    analysis = {

        'grammar': analyzer.grammar\_checker.check(text),

        'structure': analyzer.structure\_analyzer.analyze(text),

        'coherence': analyzer.coherence\_analyzer.analyze(text),

        'vocabulary': analyzer.vocabulary\_analyzer.analyze(text)

    }

    scores = analyzer.scorer.calculate\_score(text, analysis)

    analysis['scores'] = scores

    return analysis

def read\_essay\_from\_file(file\_path):

    try:

        with open(file\_path, 'r', encoding='utf-8') as file:

            return file.read()

    except Exception as e:

        print(f"Error reading file: {e}")

        return None

def main():

    print("Welcome to Essay Analyzer!")

    print("\n1. Enter essay manually")

    print("2. Upload essay from file")

    print("3. Exit")

    choice = input("\nEnter your choice (1, 2, or 3): ")

    if choice == '1':

        print("\nEnter your essay (type 'END' on a new line when finished):")

        lines = []

        while True:

            line = input()

            if line.strip().upper() == 'END':

                break

            lines.append(line)

        essay = '\n'.join(lines)

    elif choice == '2':

        file\_path = input("\nEnter the path to your essay file: ")

        essay = read\_essay\_from\_file(file\_path)

        if not essay:

            print("Failed to read essay file.")

            return

    elif choice == '3':

        print("Goodbye!")

        return

    else:

        print("Invalid choice!")

        return

    results = analyze\_essay(essay)

    print("\n=== Analysis Results ===")

    print("\nGrammar Check:")

    for error in results['grammar']:

        print(f"- {error}")

    print("\nStructure Analysis:")

    for key, value in results['structure'].items():

        print(f"- {key}: {value}")

    print("\nCoherence Analysis:")

    for key, value in results['coherence'].items():

        print(f"- {key}: {value}")

    print("\nVocabulary Analysis:")

    for key, value in results['vocabulary'].items():

        if key == 'most\_used\_words':

            print("\nMost Used Words:")

            for word, count in value:

                print(f"- '{word}': {count} times")

        else:

            print(f"- {key}: {value}")

    print("\nScores:")

    for component, score in results['scores'].items():

        print(f"{component.capitalize()}: {score:.2f}")

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

    main()