Scenario 1: Data ValidationTask: Write a function validate\_data(data) that checks if a list of dictionaries (e.g., [{"name": "Alice", "age": 30}, {"name": "Bob", "age": "25"}]) contains valid integer values for the "age" key. Return a list of invalid enteries

Answer

import typing

def validate\_data(data: typing.List[typing.Dict]) -> typing.List[typing.Dict]:

    """

    Checks if a list of dictionaries contains valid integer values for the "age" key.

    Args:

        data: A list of dictionaries, where each dictionary is expected to have an 'age' key.

    Returns:

        A new list containing only the dictionaries with invalid 'age' entries.

        An entry is considered invalid if the 'age' key is missing or its value

        cannot be converted to an integer.

    """

    # Create an empty list to store the dictionaries with invalid age entries.

    invalid\_entries = []

    # Iterate through each dictionary in the input list.

    for entry in data:

        # Use a try-except block to gracefully handle potential errors.

        try:

            # Check if the 'age' key exists in the dictionary.

            if "age" in entry:

                # Attempt to convert the value of the 'age' key to an integer.

                # This will raise a ValueError if the value is not a number.

                int(entry["age"])

            else:

                # If the 'age' key is not present, the entry is considered invalid.

                invalid\_entries.append(entry)

        except (ValueError, TypeError):

            # If a ValueError or TypeError occurs, it means the age value

            # is not a valid integer. Append the whole dictionary to the

            # list of invalid entries.

            invalid\_entries.append(entry)

    # Return the list of invalid entries found.

    return invalid\_entries

# --- Example Usage ---

# A sample list of dictionaries with mixed valid and invalid data.

sample\_data = [

    {"name": "Alice", "age": 30},

    {"name": "Bob", "age": "25"},      # Valid, as "25" can be converted to an integer.

    {"name": "Charlie", "age": "thirty"}, # Invalid, as "thirty" is a string, not a number.

    {"name": "Diana", "age": 28.5},    # Invalid, as 28.5 is a float, not an integer.

    {"name": "Eve", "age": None},      # Invalid, as None is not a number.

    {"name": "Frank"},                 # Invalid, as the 'age' key is missing.

    {"name": "Grace", "age": 42}

]

# Call the function with the sample data.

validation\_results = validate\_data(sample\_data)

# Print the results to the console.

print("Original Data:")

for item in sample\_data:

    print(item)

print("\nInvalid Entries Found:")

if validation\_results:

    for item in validation\_results:

        print(item)

else:

    print("No invalid entries were found.")

Scenario 2: Logging DecoratorTask: Create a decorator @log\_execution\_time that logs the time taken to execute a function. Use it to log the runtime of a sample function calculate\_sum(n) that returns the sum of numbers from 1 to n.

Answer

import time

import typing

def log\_execution\_time(func: typing.Callable) -> typing.Callable:

    """

    A decorator that logs the execution time of the function it wraps.

    Args:

        func: The function to be decorated.

    Returns:

        The wrapper function that logs the execution time and then calls

        the original function.

    """

    def wrapper(\*args: typing.Any, \*\*kwargs: typing.Any) -> typing.Any:

        """

        The wrapper function that contains the logging logic.

        """

        # Record the start time using a high-resolution performance counter.

        start\_time = time.perf\_counter()

        # Call the original function with its arguments.

        result = func(\*args, \*\*kwargs)

        # Record the end time.

        end\_time = time.perf\_counter()

        # Calculate the total execution time.

        run\_time = end\_time - start\_time

        # Print a formatted log message.

        print(f"Function '{func.\_\_name\_\_}' took {run\_time:.4f} seconds to execute.")

        # Return the result of the original function.

        return result

    # Return the wrapper function to replace the original function.

    return wrapper

@log\_execution\_time

def calculate\_sum(n: int) -> int:

    """

    A sample function that calculates the sum of numbers from 1 to n.

    This function is decorated to log its execution time.

    Args:

        n: The upper limit for the sum calculation.

    Returns:

        The sum of numbers from 1 to n.

    """

    total = 0

    for i in range(1, n + 1):

        total += i

    return total

# --- Example Usage ---

# Call the decorated function with a large number to see the log output.

print(f"Calculating the sum of numbers up to 10,000,000...")

sum\_result = calculate\_sum(10000000)

print(f"\nResult of the sum is: {sum\_result}")

Scenario 3: Missing Value Handling

Task: A dataset has missing values in the "income" column. Write code to:

1. Replace missing values with the median if the data is normally distributed.

2. Replace with the mode if skewed.

Use Pandas and a skewness threshold of 0.5.

5 points

Answer

import pandas as pd

import numpy as np

def impute\_missing\_values(df: pd.DataFrame, column: str, threshold: float = 0.5) -> pd.DataFrame:

    """

    Imputes missing values in a DataFrame column based on its skewness.

    If the data is normally distributed (skewness < threshold), missing values are

    replaced with the median. If the data is skewed (skewness >= threshold),

    missing values are replaced with the mode.

    Args:

        df: The pandas DataFrame containing the data.

        column: The name of the column to perform imputation on.

        threshold: The skewness threshold to determine distribution type. Default is 0.5.

    Returns:

        The DataFrame with missing values in the specified column imputed.

    """

    # Create a copy of the DataFrame to avoid modifying the original.

    df\_imputed = df.copy()

    # Calculate the skewness of the specified column.

    # The .skew() method handles NaN values by default, so we don't need to drop them.

    skewness = df\_imputed[column].skew()

    print(f"Skewness of '{column}' column is: {skewness:.4f}")

    # Check if the column is normally distributed based on the threshold.

    if abs(skewness) < threshold:

        # If normally distributed, replace missing values with the median.

        median\_value = df\_imputed[column].median()

        df\_imputed[column].fillna(median\_value, inplace=True)

        print(f"Data is normally distributed. Missing values replaced with median: {median\_value}")

    else:

        # If skewed, replace missing values with the mode.

        # .mode() returns a Series, so we take the first value [0].

        mode\_value = df\_imputed[column].mode()[0]

        df\_imputed[column].fillna(mode\_value, inplace=True)

        print(f"Data is skewed. Missing values replaced with mode: {mode\_value}")

    return df\_imputed

# --- Example Usage ---

# 1. Create a sample DataFrame with missing values and a skewed distribution.

# We'll use a positive skew to demonstrate the "mode" case.

print("--- Scenario: Skewed Data ---")

data\_skewed = {

    'income': [50000, 52000, 55000, 60000, 100000, np.nan, 200000, 58000, 61000],

    'city': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']

}

df\_skewed = pd.DataFrame(data\_skewed)

print("Original DataFrame (Skewed):")

print(df\_skewed)

# Impute missing values for the skewed dataset.

df\_skewed\_imputed = impute\_missing\_values(df\_skewed, 'income', threshold=0.5)

print("\nDataFrame After Imputation (Skewed):")

print(df\_skewed\_imputed)

print("\n" + "="\*50 + "\n")

# 2. Create a sample DataFrame with missing values and a more normal distribution.

# We'll make the data more balanced to demonstrate the "median" case.

print("--- Scenario: Normally Distributed Data ---")

data\_normal = {

    'income': [55000, 58000, 60000, 62000, 65000, np.nan, 68000, 70000, 75000],

    'city': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']

}

df\_normal = pd.DataFrame(data\_normal)

print("Original DataFrame (Normally Distributed):")

print(df\_normal)

# Impute missing values for the normally distributed dataset.

df\_normal\_imputed = impute\_missing\_values(df\_normal, 'income', threshold=0.5)

print("\nDataFrame After Imputation (Normally Distributed):")

print(df\_normal\_imputed)

Scenario 4: Text Pre-processing

Task: Clean a text column in a DataFrame by:

1. Converting to lowercase.

2. Removing special characters (e.g., !, @).

3. Tokenizing the text.

Answer

import pandas as pd

import re

import typing

def preprocess\_text\_column(df: pd.DataFrame, column: str) -> pd.DataFrame:

    """

    Cleans a text column in a DataFrame by converting to lowercase, removing

    special characters, and tokenizing the text.

    Args:

        df: The pandas DataFrame containing the text data.

        column: The name of the column to be pre-processed.

    Returns:

        The DataFrame with a new column '\_processed' containing the cleaned and

        tokenized text.

    """

    # Create a copy of the DataFrame to avoid modifying the original.

    df\_processed = df.copy()

    # Step 1: Convert the text to lowercase.

    # We apply the .str.lower() method to the specified column.

    df\_processed[column] = df\_processed[column].str.lower()

    # Step 2: Remove special characters.

    # We use a regular expression to replace any characters that are not

    # alphanumeric or whitespace with a blank space.

    df\_processed['processed\_text'] = df\_processed[column].apply(

        lambda x: re.sub(r'[^a-z0-9\s]', '', x) if isinstance(x, str) else x

    )

    # Step 3: Tokenize the text.

    # We use the .str.split() method to split the string into a list of words.

    df\_processed['processed\_text'] = df\_processed['processed\_text'].str.split()

    return df\_processed

# --- Example Usage ---

# Create a sample DataFrame with a text column that needs cleaning.

sample\_data = {

    'product\_review': [

        "This product is AWESOME! I love it.",

        "It's okay... but the price is too high.",

        "I'm very disappointed!!!",

        "Fast shipping & great quality!!!",

        "Just received it, haven't used yet."

    ]

}

df = pd.DataFrame(sample\_data)

print("Original DataFrame:")

print(df)

print("\n" + "="\*50 + "\n")

# Call the text pre-processing function.

df\_clean = preprocess\_text\_column(df, 'product\_review')

print("DataFrame After Pre-processing:")

print(df\_clean)

Scenario 5: Hyperparameter Tuning

Task: Use GridSearchCV to find the best max\_depth (values: [3, 5, 7]) and n\_estimators (values: [50, 100]) for a Random Forest classifier.

5 points

import pandas as pd

import numpy as np

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

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification\_report

from sklearn.datasets import make\_classification

import typing

def perform\_grid\_search(X: pd.DataFrame, y: pd.Series) -> None:

    """

    Performs a grid search to find the best hyperparameters for a Random Forest Classifier.

    This function uses GridSearchCV to systematically work through multiple

    combinations of hyperparameter values, cross-validating the results to

    determine which combination works best.

    Args:

        X: The feature data (pandas DataFrame).

        y: The target labels (pandas Series).

    """

    # 1. Split the data into training and testing sets.

    # We use a 70/30 split to train the model and then evaluate its performance.

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

        X, y, test\_size=0.3, random\_state=42

    )

    print("Data split into training and testing sets.")

    # 2. Define the model and the parameter grid to search.

    # The grid contains the hyperparameter values we want to test.

    # Here, we will search for the best combination of `max\_depth` and `n\_estimators`.

    model = RandomForestClassifier(random\_state=42)

    param\_grid = {

        'n\_estimators': [50, 100],  # The number of trees in the forest.

        'max\_depth': [3, 5, 7]      # The maximum depth of each tree.

    }

    print("Model and parameter grid defined.")

    # 3. Instantiate GridSearchCV.

    # We create an instance of GridSearchCV, passing the model, the parameter grid,

    # the scoring metric to optimize ('accuracy'), and the number of cross-validation

    # folds (cv=5).

    grid\_search = GridSearchCV(

        estimator=model,

        param\_grid=param\_grid,

        scoring='accuracy',

        cv=5,

        n\_jobs=-1,  # Use all available CPU cores for faster processing.

        verbose=1

    )

    print("GridSearchCV instantiated.")

    # 4. Fit the grid search to the training data.

    # This process will train a model for each combination of parameters and

    # evaluate them using cross-validation.

    print("\nStarting grid search...")

    grid\_search.fit(X\_train, y\_train)

    # 5. Print the results.

    print("\nGrid Search Complete!")

    print("="\*50)

    print("Best parameters found: ", grid\_search.best\_params\_)

    print("Best cross-validation score: {:.2f}".format(grid\_search.best\_score\_))

    print("="\*50)

    # 6. Evaluate the best model on the test set.

    # We can now use the best model found by the grid search to make predictions

    # on our unseen test data.

    best\_model = grid\_search.best\_estimator\_

    y\_pred = best\_model.predict(X\_test)

    print("\nClassification Report on the test set:")

    print(classification\_report(y\_test, y\_pred))

# --- Example Usage ---

# Generate a synthetic dataset for demonstration.

# This creates a dataset with 1000 samples, 10 features, and 2 classes.

X, y = make\_classification(

    n\_samples=1000,

    n\_features=10,

    n\_classes=2,

    random\_state=42

)

# Convert the NumPy arrays to pandas DataFrame and Series for clarity.

X\_df = pd.DataFrame(X, columns=[f'feature\_{i}' for i in range(X.shape[1])])

y\_series = pd.Series(y, name='target')

print("Sample dataset created with 1000 records.")

# Run the grid search function with the sample data.

perform\_grid\_search(X\_df, y\_series)

Scenario 6: Custom Evaluation Metric

Task: Implement a custom metric weighted\_accuracy where class 0 has a weight of 1 and class 1 has a weight of 2.

5 points

import pandas as pd

import numpy as np

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

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification\_report, make\_scorer, accuracy\_score

from sklearn.datasets import make\_classification

import typing

def weighted\_accuracy\_score(y\_true: np.ndarray, y\_pred: np.ndarray) -> float:

    """

    Calculates a weighted accuracy score where class 1 is weighted more heavily

    than class 0.

    Args:

        y\_true: True labels of the data.

        y\_pred: Predicted labels from the model.

    Returns:

        The weighted accuracy score.

    """

    # Identify the indices for each class

    class\_0\_indices = y\_true == 0

    class\_1\_indices = y\_true == 1

    # Calculate accuracy for each class separately

    accuracy\_class\_0 = accuracy\_score(y\_true[class\_0\_indices], y\_pred[class\_0\_indices])

    accuracy\_class\_1 = accuracy\_score(y\_true[class\_1\_indices], y\_pred[class\_1\_indices])

    # Define weights for each class

    weight\_class\_0 = 1.0

    weight\_class\_1 = 2.0

    # Calculate the weighted average of the accuracies

    weighted\_score = (accuracy\_class\_0 \* weight\_class\_0 + accuracy\_class\_1 \* weight\_class\_1) / (weight\_class\_0 + weight\_class\_1)

    return weighted\_score

# Create a scorer object from the custom function

weighted\_accuracy\_scorer = make\_scorer(weighted\_accuracy\_score)

def perform\_grid\_search(X: pd.DataFrame, y: pd.Series) -> None:

    """

    Performs a grid search to find the best hyperparameters for a Random Forest Classifier

    using a custom weighted accuracy metric.

    This function uses GridSearchCV to systematically work through multiple

    combinations of hyperparameter values, cross-validating the results to

    determine which combination works best.

    Args:

        X: The feature data (pandas DataFrame).

        y: The target labels (pandas Series).

    """

    # 1. Split the data into training and testing sets.

    # We use a 70/30 split to train the model and then evaluate its performance.

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

        X, y, test\_size=0.3, random\_state=42

    )

    print("Data split into training and testing sets.")

    # 2. Define the model and the parameter grid to search.

    # The grid contains the hyperparameter values we want to test.

    # Here, we will search for the best combination of `max\_depth` and `n\_estimators`.

    model = RandomForestClassifier(random\_state=42)

    param\_grid = {

        'n\_estimators': [50, 100],  # The number of trees in the forest.

        'max\_depth': [3, 5, 7]      # The maximum depth of each tree.

    }

    print("Model and parameter grid defined.")

    # 3. Instantiate GridSearchCV.

    # We create an instance of GridSearchCV, passing the model, the parameter grid,

    # the scoring metric to optimize (our custom weighted\_accuracy), and the number

    # of cross-validation folds (cv=5).

    grid\_search = GridSearchCV(

        estimator=model,

        param\_grid=param\_grid,

        scoring=weighted\_accuracy\_scorer,

        cv=5,

        n\_jobs=-1,  # Use all available CPU cores for faster processing.

        verbose=1

    )

    print("GridSearchCV instantiated with custom weighted accuracy score.")

    # 4. Fit the grid search to the training data.

    # This process will train a model for each combination of parameters and

    # evaluate them using cross-validation.

    print("\nStarting grid search...")

    grid\_search.fit(X\_train, y\_train)

    # 5. Print the results.

    print("\nGrid Search Complete!")

    print("="\*50)

    print("Best parameters found: ", grid\_search.best\_params\_)

    print("Best cross-validation score: {:.2f}".format(grid\_search.best\_score\_))

    print("="\*50)

    # 6. Evaluate the best model on the test set.

    # We can now use the best model found by the grid search to make predictions

    # on our unseen test data.

    best\_model = grid\_search.best\_estimator\_

    y\_pred = best\_model.predict(X\_test)

    print("\nClassification Report on the test set:")

    print(classification\_report(y\_test, y\_pred))

# --- Example Usage ---

# Generate a synthetic dataset for demonstration.

# This creates a dataset with 1000 samples, 10 features, and 2 classes.

X, y = make\_classification(

    n\_samples=1000,

    n\_features=10,

    n\_classes=2,

    random\_state=42

)

# Convert the NumPy arrays to pandas DataFrame and Series for clarity.

X\_df = pd.DataFrame(X, columns=[f'feature\_{i}' for i in range(X.shape[1])])

y\_series = pd.Series(y, name='target')

print("Sample dataset created with 1000 records.")

# Run the grid search function with the sample data.

perform\_grid\_search(X\_df, y\_series)

Scenario 7: Image Augmentation

Task: Use TensorFlow/Keras to create an image augmentation pipeline with random rotations (±20 degrees), horizontal flips, and zoom (0.2x).

5 points

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import typing

def display\_images(original\_images: np.ndarray, augmented\_images: np.ndarray) -> None:

    """

    Displays a set of original and augmented images side by side.

    Args:

        original\_images: A batch of original images as a NumPy array.

        augmented\_images: A batch of augmented images as a NumPy array.

    """

    # Create a figure to hold the plots.

    plt.figure(figsize=(10, 5))

    num\_images = original\_images.shape[0]

    # Iterate through the images to display them in a grid.

    for i in range(num\_images):

        # Display the original image in the first row.

        plt.subplot(2, num\_images, i + 1)

        plt.imshow(original\_images[i])

        plt.title("Original")

        plt.axis("off")

        # Display the augmented image in the second row.

        plt.subplot(2, num\_images, i + 1 + num\_images)

        plt.imshow(augmented\_images[i])

        plt.title("Augmented")

        plt.axis("off")

    plt.tight\_layout()

    plt.show()

def create\_augmentation\_pipeline() -> tf.keras.Sequential:

    """

    Creates a TensorFlow/Keras Sequential model for image augmentation.

    The pipeline includes random rotations, horizontal flips, and random zoom.

    Returns:

        A tf.keras.Sequential model representing the augmentation pipeline.

    """

    # Use a tf.keras.Sequential model to chain the augmentation layers.

    # The layers will be applied to the input data in the order they are added.

    data\_augmentation = tf.keras.Sequential([

        # Randomly rotate images by a factor of up to 0.2 (which is 20% of 2\*pi radians, or 360 degrees).

        # This corresponds to a range of +/- 72 degrees.

        # For a rotation of +/- 20 degrees, the factor should be 20/360 \* 2\*pi, which is approximately 0.349.

        # Let's use the provided 20 degrees. The `factor` argument is a float that specifies the upper bound for the rotation angle in radians.

        tf.keras.layers.RandomRotation(factor=20 \* (np.pi/180), seed=42),

        # Randomly flip images horizontally.

        tf.keras.layers.RandomFlip("horizontal", seed=42),

        # Randomly zoom in on images by a factor of 0.2.

        # The `height\_factor` and `width\_factor` can be a single float.

        tf.keras.layers.RandomZoom(height\_factor=0.2, seed=42),

    ])

    return data\_augmentation

# --- Example Usage ---

# Create a batch of 8 sample images with a size of 128x128 pixels and 3 color channels.

# These will serve as our dummy dataset to test the pipeline.

# The pixel values are between 0 and 1, as expected by Keras layers.

sample\_images = np.random.rand(8, 128, 128, 3)

# Create the augmentation pipeline model.

augmentation\_pipeline = create\_augmentation\_pipeline()

# Apply the pipeline to the sample images.

# The `call` method of the Keras model applies the transformations.

augmented\_images = augmentation\_pipeline(sample\_images, training=True)

# Convert the TensorFlow tensor to a NumPy array for plotting.

augmented\_images\_np = augmented\_images.numpy()

# Display the original and augmented images.

print("Displaying original vs. augmented images...")

display\_images(sample\_images, augmented\_images\_np)

Scenario 8: Model Callbacks

Task: Implement an EarlyStopping callback that stops training if validation loss doesn’t improve for 3 epochs and restores the best weights.

Answer

import tensorflow as tf

from tensorflow import keras

import numpy as np

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

from sklearn.datasets import make\_classification

import typing

def build\_simple\_model() -> keras.Sequential:

    """

    Builds a simple Keras Sequential model for a classification task.

    This function defines a neural network with three dense layers and a final

    output layer with a sigmoid activation function, suitable for binary classification.

    Returns:

        A compiled Keras Sequential model.

    """

    # Define a simple Sequential model.

    model = keras.Sequential([

        # The input layer with 32 neurons and ReLU activation.

        keras.layers.Dense(32, activation='relu', input\_shape=(10,)),

        # An intermediate layer with 16 neurons.

        keras.layers.Dense(16, activation='relu'),

        # The output layer with a single neuron and a sigmoid activation for binary classification.

        keras.layers.Dense(1, activation='sigmoid')

    ])

    # Compile the model with an optimizer, a loss function, and a metric.

    model.compile(optimizer='adam',

                  loss='binary\_crossentropy',

                  metrics=['accuracy'])

    return model

def train\_with\_early\_stopping(model: keras.Sequential, X: np.ndarray, y: np.ndarray) -> None:

    """

    Trains a Keras model on a dataset using an EarlyStopping callback.

    The callback will monitor the validation loss and stop training if it does not

    improve for a specified number of epochs, restoring the best weights found.

    Args:

        model: The compiled Keras model to be trained.

        X: The feature data (NumPy array).

        y: The target labels (NumPy array).

    """

    # Split the data into training and validation sets.

    # The validation set is crucial for the EarlyStopping callback to work correctly.

    X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

    # Instantiate the EarlyStopping callback.

    # The `monitor` argument specifies the metric to watch (validation loss).

    # The `patience` argument defines the number of epochs with no improvement

    # after which training will be stopped.

    # `restore\_best\_weights` ensures the model reverts to the epoch with the lowest

    # validation loss.

    early\_stopping\_callback = keras.callbacks.EarlyStopping(

        monitor='val\_loss',

        patience=3,

        restore\_best\_weights=True

    )

    # Train the model, passing the callback in the 'callbacks' list.

    print("Starting model training. The process will stop early if the validation loss plateaus...")

    history = model.fit(

        X\_train,

        y\_train,

        epochs=100,  # A large number of epochs to allow for early stopping to occur.

        validation\_data=(X\_val, y\_val),

        callbacks=[early\_stopping\_callback],

        verbose=1  # Show progress during training.

    )

    print("\nTraining complete.")

    print(f"Training stopped after {len(history.history['loss'])} epochs.")

    print("The model has been restored to the best weights found during training.")

# --- Example Usage ---

# 1. Generate a synthetic dataset for a binary classification problem.

# This makes the example fully runnable without needing an external dataset.

X, y = make\_classification(

    n\_samples=500,

    n\_features=10,

    n\_informative=5,

    n\_redundant=0,

    random\_state=42

)

# 2. Build the model.

model = build\_simple\_model()

# 3. Train the model using the EarlyStopping callback.

train\_with\_early\_stopping(model, X, y)

Scenario 9: Structured Response Generation

Task: Use the Gemini API to generate a response in JSON format for the query: "List 3 benefits of Python for data science." Handle cases where the response isn’t valid JSON.

Answer

import { useState } from 'react';

import { Sparkles, Loader2, RefreshCw } from 'lucide-react';

// Main application component

const App = () => {

  const [response, setResponse] = useState('');

  const [loading, setLoading] = useState(false);

  const [error, setError] = useState('');

  // Function to call the Gemini API and get a structured JSON response

  const generateStructuredResponse = async () => {

    setLoading(true);

    setError('');

    setResponse('');

    // The user's prompt for the model

    const prompt = "List 3 benefits of Python for data science.";

    // Define the JSON schema for the desired response format

    const responseSchema = {

      type: "ARRAY",

      items: {

        type: "OBJECT",

        properties: {

          "benefit": { "type": "STRING" }

        },

        "propertyOrdering": ["benefit"]

      }

    };

    // Construct the payload for the API call

    const payload = {

      contents: [{

        parts: [{ text: prompt }]

      }],

      generationConfig: {

        responseMimeType: "application/json",

        responseSchema: responseSchema

      }

    };

    const apiKey = "";

    const apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-flash-preview-05-20:generateContent?key=${apiKey}`;

    // Use exponential backoff for API calls

    const maxRetries = 5;

    let retries = 0;

    while (retries < maxRetries) {

      try {

        const fetchResponse = await fetch(apiUrl, {

          method: 'POST',

          headers: { 'Content-Type': 'application/json' },

          body: JSON.stringify(payload)

        });

        if (!fetchResponse.ok) {

          throw new Error(`API error: ${fetchResponse.statusText}`);

        }

        const result = await fetchResponse.json();

        // Check if the response structure is valid and extract the JSON string

        if (

          result.candidates &&

          result.candidates.length > 0 &&

          result.candidates[0].content &&

          result.candidates[0].content.parts &&

          result.candidates[0].content.parts.length > 0

        ) {

          const jsonString = result.candidates[0].content.parts[0].text;

          try {

            // Attempt to parse the JSON string to validate it

            const parsedJson = JSON.parse(jsonString);

            // Store the pretty-printed JSON for display

            setResponse(JSON.stringify(parsedJson, null, 2));

            setLoading(false);

            return;

          } catch (e) {

            // Handle cases where the response text is not valid JSON

            throw new Error('Invalid JSON format received from the API.');

          }

        } else {

          throw new Error('Unexpected API response structure.');

        }

      } catch (e) {

        if (fetchResponse && fetchResponse.status === 429 && retries < maxRetries - 1) {

          const delay = Math.pow(2, retries) \* 1000;

          await new Promise(res => setTimeout(res, delay));

          retries++;

        } else {

          console.error('Error generating response:', e);

          setError(`Failed to generate a valid response: ${e.message}`);

          break; // Exit the loop on non-retriable errors

        }

      }

    }

    setLoading(false);

  };

  return (

    <div className="min-h-screen bg-gray-900 text-white p-4 sm:p-8 flex flex-col items-center">

      <div className="w-full max-w-2xl bg-gray-800 p-6 sm:p-8 rounded-2xl shadow-lg border border-gray-700">

        <h1 className="text-3xl sm:text-4xl font-extrabold text-center text-teal-400 mb-2">

          Structured Response Generator

        </h1>

        <p className="text-center text-gray-400 mb-6 sm:mb-8">

          Get a JSON response from the Gemini API.

        </p>

        <div className="flex flex-col sm:flex-row items-center justify-center space-y-4 sm:space-y-0 sm:space-x-4 mb-8">

          <button

            onClick={generateStructuredResponse}

            disabled={loading}

            className="w-full sm:w-auto px-6 py-3 bg-blue-600 hover:bg-blue-700 disabled:bg-blue-800 text-white rounded-full font-semibold transition-colors duration-200 flex items-center justify-center space-x-2 shadow-lg"

          >

            {loading ? (

              <>

                <Loader2 className="animate-spin" size={20} />

                <span>Generating...</span>

              </>

            ) : (

              <>

                <Sparkles size={20} />

                <span>Generate JSON</span>

              </>

            )}

          </button>

          {loading && (

            <div className="flex items-center space-x-2 text-gray-400">

              <RefreshCw className="animate-spin" size={16} />

              <p>Waiting for model response...</p>

            </div>

          )}

        </div>

        {error && (

          <div className="bg-red-900 text-red-300 p-4 rounded-xl border border-red-700 text-sm">

            <p className="font-semibold mb-2">Error:</p>

            <p>{error}</p>

            <p className="mt-2 text-gray-400 text-xs">This may be due to an API issue or an invalid JSON response.</p>

          </div>

        )}

        {response && (

          <div className="bg-gray-700 rounded-xl p-4 overflow-x-auto border border-gray-600 shadow-inner">

            <h3 className="text-xl font-semibold text-teal-300 mb-2">Generated JSON:</h3>

            <pre className="text-green-300 whitespace-pre-wrap break-all text-sm leading-relaxed">

              {response}

            </pre>

          </div>

        )}

      </div>

    </div>

  );

};

export default App;

Scenario 10: Summarization with Constraints

Task: Write a prompt to summarize a news article into 2 sentences. If the summary exceeds 50 words, truncate it to the nearest complete sentence

import { useState } from 'react';

import { Sparkles, Loader2, RefreshCw, FileText, Code } from 'lucide-react';

// Main application component

const App = () => {

  const [response, setResponse] = useState('');

  const [loading, setLoading] = useState(false);

  const [error, setError] = useState('');

  const [mode, setMode] = useState('json'); // 'json' or 'summarization'

  const [articleText, setArticleText] = useState('');

  const [summary, setSummary] = useState('');

  // Function to handle the API call with exponential backoff

  const callApiWithBackoff = async (payload, apiUrl, mimeType) => {

    const maxRetries = 5;

    let retries = 0;

    while (retries < maxRetries) {

      try {

        const fetchResponse = await fetch(apiUrl, {

          method: 'POST',

          headers: { 'Content-Type': 'application/json' },

          body: JSON.stringify(payload)

        });

        if (!fetchResponse.ok) {

          if (fetchResponse.status === 429) {

            throw new Error('API throttling.');

          }

          throw new Error(`API error: ${fetchResponse.statusText}`);

        }

        const result = await fetchResponse.json();

        // Check for valid response structure

        if (

          result.candidates &&

          result.candidates.length > 0 &&

          result.candidates[0].content &&

          result.candidates[0].content.parts &&

          result.candidates[0].content.parts.length > 0

        ) {

          if (mimeType === 'application/json') {

            const jsonString = result.candidates[0].content.parts[0].text;

            try {

              return JSON.parse(jsonString);

            } catch (e) {

              throw new Error('Invalid JSON format received from the API.');

            }

          } else {

            return result.candidates[0].content.parts[0].text;

          }

        } else {

          throw new Error('Unexpected API response structure.');

        }

      } catch (e) {

        if (e.message === 'API throttling.' && retries < maxRetries - 1) {

          const delay = Math.pow(2, retries) \* 1000;

          await new Promise(res => setTimeout(res, delay));

          retries++;

        } else {

          throw e; // Re-throw non-retriable or final-attempt errors

        }

      }

    }

    throw new Error('Max retries exceeded.');

  };

  // Function to handle JSON generation

  const generateStructuredResponse = async () => {

    setLoading(true);

    setError('');

    setResponse('');

    const prompt = "List 3 benefits of Python for data science.";

    const responseSchema = {

      type: "ARRAY",

      items: {

        type: "OBJECT",

        properties: { "benefit": { "type": "STRING" } },

        "propertyOrdering": ["benefit"]

      }

    };

    const payload = {

      contents: [{ parts: [{ text: prompt }] }],

      generationConfig: {

        responseMimeType: "application/json",

        responseSchema: responseSchema

      }

    };

    const apiKey = "";

    const apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-flash-preview-05-20:generateContent?key=${apiKey}`;

    try {

      const parsedJson = await callApiWithBackoff(payload, apiUrl, 'application/json');

      setResponse(JSON.stringify(parsedJson, null, 2));

    } catch (e) {

      setError(`Failed to generate a valid response: ${e.message}`);

    } finally {

      setLoading(false);

    }

  };

  // Function to handle summarization

  const handleSummarization = async () => {

    setLoading(true);

    setError('');

    setSummary('');

    if (!articleText.trim()) {

      setError('Please enter a news article to summarize.');

      setLoading(false);

      return;

    }

    const prompt = `Summarize the following news article in 2 concise sentences. The summary should not exceed 50 words. If the word count is exceeded, please truncate the summary to the nearest complete sentence that keeps the word count under the limit.\n\nArticle: ${articleText}`;

    const payload = {

      contents: [{ parts: [{ text: prompt }] }],

      generationConfig: {}

    };

    const apiKey = "";

    const apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-flash-preview-05-20:generateContent?key=${apiKey}`;

    try {

      let fullSummary = await callApiWithBackoff(payload, apiUrl, 'text/plain');

      // Post-processing to enforce constraints

      const sentences = fullSummary.match(/[^.!?]+[.!?]/g) || [fullSummary];

      let finalSummary = '';

      let wordCount = 0;

      for (let i = 0; i < Math.min(sentences.length, 2); i++) {

        const sentence = sentences[i].trim();

        const sentenceWordCount = sentence.split(/\s+/).length;

        if (wordCount + sentenceWordCount <= 50) {

          finalSummary += (finalSummary ? ' ' : '') + sentence;

          wordCount += sentenceWordCount;

        } else {

          break; // Stop adding sentences if the word count is exceeded

        }

      }

      setSummary(finalSummary || fullSummary); // Fallback to full summary if truncation fails

    } catch (e) {

      setError(`Failed to generate a summary: ${e.message}`);

    } finally {

      setLoading(false);

    }

  };

  const isJsonMode = mode === 'json';

  return (

    <div className="min-h-screen bg-gray-900 text-white p-4 sm:p-8 flex flex-col items-center">

      <div className="w-full max-w-3xl bg-gray-800 p-6 sm:p-8 rounded-2xl shadow-lg border border-gray-700">

        <h1 className="text-3xl sm:text-4xl font-extrabold text-center text-teal-400 mb-2">

          Multifunctional Gemini App

        </h1>

        <p className="text-center text-gray-400 mb-6 sm:mb-8">

          Switch between generating structured JSON or summarizing articles.

        </p>

        <div className="flex justify-center space-x-4 mb-8">

          <button

            onClick={() => setMode('json')}

            className={`px-4 py-2 rounded-full font-semibold transition-colors duration-200 flex items-center space-x-2 ${

              isJsonMode ? 'bg-blue-600 text-white' : 'bg-gray-700 text-gray-300 hover:bg-gray-600'

            }`}

          >

            <Code size={18} />

            <span>JSON Generator</span>

          </button>

          <button

            onClick={() => setMode('summarization')}

            className={`px-4 py-2 rounded-full font-semibold transition-colors duration-200 flex items-center space-x-2 ${

              !isJsonMode ? 'bg-blue-600 text-white' : 'bg-gray-700 text-gray-300 hover:bg-gray-600'

            }`}

          >

            <FileText size={18} />

            <span>Summarizer</span>

          </button>

        </div>

        {isJsonMode ? (

          <div>

            <div className="flex flex-col sm:flex-row items-center justify-center space-y-4 sm:space-y-0 sm:space-x-4 mb-8">

              <button

                onClick={generateStructuredResponse}

                disabled={loading}

                className="w-full sm:w-auto px-6 py-3 bg-blue-600 hover:bg-blue-700 disabled:bg-blue-800 text-white rounded-full font-semibold transition-colors duration-200 flex items-center justify-center space-x-2 shadow-lg"

              >

                {loading ? (

                  <>

                    <Loader2 className="animate-spin" size={20} />

                    <span>Generating...</span>

                  </>

                ) : (

                  <>

                    <Sparkles size={20} />

                    <span>Generate JSON</span>

                  </>

                )}

              </button>

              {loading && (

                <div className="flex items-center space-x-2 text-gray-400">

                  <RefreshCw className="animate-spin" size={16} />

                  <p>Waiting for model response...</p>

                </div>

              )}

            </div>

            {error && (

              <div className="bg-red-900 text-red-300 p-4 rounded-xl border border-red-700 text-sm">

                <p className="font-semibold mb-2">Error:</p>

                <p>{error}</p>

                <p className="mt-2 text-gray-400 text-xs">This may be due to an API issue or an invalid JSON response.</p>

              </div>

            )}

            {response && (

              <div className="bg-gray-700 rounded-xl p-4 overflow-x-auto border border-gray-600 shadow-inner">

                <h3 className="text-xl font-semibold text-teal-300 mb-2">Generated JSON:</h3>

                <pre className="text-green-300 whitespace-pre-wrap break-all text-sm leading-relaxed">

                  {response}

                </pre>

              </div>

            )}

          </div>

        ) : (

          <div>

            <textarea

              className="w-full h-48 p-4 bg-gray-700 text-white rounded-xl resize-none focus:outline-none focus:ring-2 focus:ring-blue-500 transition-colors duration-200 mb-4"

              placeholder="Enter the news article here..."

              value={articleText}

              onChange={(e) => setArticleText(e.target.value)}

            />

            <div className="flex flex-col sm:flex-row items-center justify-center space-y-4 sm:space-y-0 sm:space-x-4 mb-8">

              <button

                onClick={handleSummarization}

                disabled={loading}

                className="w-full sm:w-auto px-6 py-3 bg-blue-600 hover:bg-blue-700 disabled:bg-blue-800 text-white rounded-full font-semibold transition-colors duration-200 flex items-center justify-center space-x-2 shadow-lg"

              >

                {loading ? (

                  <>

                    <Loader2 className="animate-spin" size={20} />

                    <span>Summarizing...</span>

                  </>

                ) : (

                  <>

                    <Sparkles size={20} />

                    <span>Summarize Article</span>

                  </>

                )}

              </button>

              {loading && (

                <div className="flex items-center space-x-2 text-gray-400">

                  <RefreshCw className="animate-spin" size={16} />

                  <p>Waiting for model response...</p>

                </div>

              )}

            </div>

            {error && (

              <div className="bg-red-900 text-red-300 p-4 rounded-xl border border-red-700 text-sm">

                <p className="font-semibold mb-2">Error:</p>

                <p>{error}</p>

              </div>

            )}

            {summary && (

              <div className="bg-gray-700 rounded-xl p-4 border border-gray-600 shadow-inner">

                <h3 className="text-xl font-semibold text-teal-300 mb-2">Generated Summary:</h3>

                <p className="text-white whitespace-pre-wrap leading-relaxed">

                  {summary}

                </p>

              </div>

            )}

          </div>

        )}

      </div>

    </div>

  );

};

export default App;